UNIVERSIDAD NACIONAL DE CAAGUAZÚ
FACULTAD DE CIENCIAS Y TECNOLOGÍAS
CARRERA DE INGENIERÍA EN ELECTRÓNICA

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA AUTOMATIZADO PARA EL MONITOREO, CONTROL Y CORRECCIÓN EN TIEMPO REAL DE PARÁMETROS CRÍTICOS EN LOS ESTANQUES DE PISCICULTURA DE LA FACULTAD DE CIENCIAS DE LA PRODUCCIÓN DE LA UNIVERSIDAD NACIONAL DE CAAGUAZÚ

Abel Duré Mendez

Fredy Daniel López Mendoza

Tutor: Ing. Antonio Zorrilla

CORONEL OVIEDO, DICIEMBRE DE 2025

DERECHO DE AUTOR

Quienes suscriben, **Fredy López** y **Abel Duré**, autor/a/autores del trabajo de investigación titulado "**Diseño e implementación de un sistema automatizado para el monitoreo, control y corrección en tiempo real de parámetros críticos en los estanques de piscicultura de la Facultad de Ciencias de la producción de la Universidad Nacional de Caaguazú**", declaran que voluntariamente ceden a título gratuito en forma pura y simple ilimitada e irrevocablemente a favor de la Facultad de Ciencias y Tecnologías – UNCA, el derecho de autor de contenido patrimonial, que le corresponde sobre el trabajo de referencia. Conforme a lo anteriormente expresado, esta sesión le otorga a la FCyT la Facultad de comunicar la obra divulgarla, publicarla y reproducirla en soportes analógicos o digitales en la oportunidad que así lo estime conveniente. La FCyT deberá indicar qué autoría o creación del trabajo corresponde a mi persona y hará referencia al autor y a las personas que hayan colaborado en la realización del presente trabajo de investigación.

En la ciudad de Coronel Oviedo a los …. , del mes de …..,,,,,,,,,,,,,, del 2025

…….

Firma/s

PÁGINA DE APROBACIÓN

Trabajo de fin de grado para la obtención del Título de Ingeniero Electricista, aprobado en representación de la Facultad Ciencias y Tecnología de la Universidad Nacional de Caaguazú, por el Tribunal Examinador constituido por los siguientes profesores y con la siguiente nota final:

CALIFICACIÓN FINAL: _____

ACTA N°:_____

FECHA :_____

Prof. Ing.

Prof. Ing.

Prof. Ing.

Dedicatoria

Dedico este trabajo a mis padres, quienes hicieron absolutamente todo para que pudiera cruzar esta meta, dejándome a mí únicamente la tarea de estudiar. Estuvieron presentes en cada tramo de este difícil proceso.

A mis amigos y compañeros de la carrera, quienes siempre estuvieron para apoyarme, ya sea aclarando dudas, compartiendo conocimientos o levantándome el ánimo cuando fue necesario

Abel Duré Mendez.

Dedicatoria

Dedico este trabajo a mi familia, por ser mi mayor pilar y brindarme siempre su apoyo incondicional.

A mi madre, cuyo sacrificio diario y amor incansable hicieron posible que yo pudiera estudiar con tranquilidad y sin que nada me faltara.

A mi abuela, por su compañía constante y por ser refugio en mis días más difíciles.

A mi novia, por su cariño, su paciencia y por recordarme siempre que la meta estaba cerca, motivándome a no rendirme y seguir adelante.

A mis compañeros y amigos de la carrera, que se convirtieron en una familia; gracias por apoyarme, aclarar dudas, compartir conocimientos y levantarme el ánimo cuando más lo necesitaba.

Y a mi gato Mike, cuya compañía silenciosa hizo más llevaderas las largas noches de estudio.

A todos ustedes, gracias por ser parte esencial de este logro.

Fredy Daniel López Mendoza.

Agradecimientos

Expresamos nuestro agradecimiento al Ing. Antonio Zorrilla por sus valiosas observaciones y por el apoyo brindado durante el desarrollo de esta obra, los cuales nos permitieron orientar adecuadamente nuestro trabajo.

Resumen

El monitoreo y control de la calidad del agua en sistemas de piscicultura es fundamental para garantizar la supervivencia, el crecimiento y la salud de los peces. En los estanques de la Facultad de Ciencias de la Producción de la Universidad Nacional del Caaguazú (UNCA), el seguimiento de parámetros críticos como el oxígeno disuelto y el pH se realizaba de forma manual y con baja frecuencia, lo que dificultaba la detección temprana de condiciones adversas y generaba pérdidas en la producción. Este proyecto tiene como objetivo diseñar e implementar un sistema automatizado para el monitoreo, control y corrección en tiempo real de los parámetros críticos de calidad del agua en los estanques de piscicultura de la institución, utilizando tecnologías embebidas y estrategias de control automático. La metodología incluyó el relevamiento de las condiciones actuales de manejo, la definición de requerimientos junto al encargado de los estanques, la selección e integración de sensores de pH, oxígeno disuelto y temperatura, así como de actuadores basados en bombas peristálticas y aireadores. Se desarrolló una arquitectura de hardware y software centrada en una Raspberry Pi 4, en la cual se implementaron controladores PID, rutinas de seguridad, registro histórico de datos y una interfaz gráfica accesible vía web para la supervisión remota del sistema. Los resultados muestran que el prototipo es capaz de medir los parámetros de interés con precisión adecuada, mantenerlos dentro de rangos recomendados mediante acciones de control oportunas y generar alarmas ante situaciones críticas. La solución propuesta contribuye a mejorar la gestión de los estanques, reducir la dependencia de la supervisión manual y fortalecer la capacidad de la UNCA para la enseñanza, la investigación aplicada y la producción acuícola.

Palabras clave: *Control y Automatización Industrial, Piscicultura, Sistemas embebidos, Calidad del agua, ODS 2 - Hambre cero, ODS 12 - Producción y Consumo Responsables.*

Abstract

Monitoring and controlling water quality in aquaculture systems is essential to ensure fish survival, growth, and health. In the ponds of the Faculty of Production Sciences at the National University of Caaguazú (UNCA), the monitoring of critical parameters such as dissolved oxygen and pH was carried out manually and with low frequency, which hindered the early detection of adverse conditions and led to production losses. This project aims to design and implement an automated system for real-time monitoring, control, and correction of critical water quality parameters in the institution's fish ponds, using embedded technologies and automatic control strategies. The methodology included surveying the current management conditions, defining requirements together with the pond manager, and selecting and integrating pH, dissolved oxygen, and temperature sensors, as well as actuators based on peristaltic pumps and aerators. A hardware and software architecture was developed around a Raspberry Pi 4, on which PID controllers, safety routines, historical data logging, and a web-based graphical interface for remote system supervision were implemented. The results show that the prototype is capable of measuring the relevant parameters with adequate accuracy, maintaining them within recommended ranges through timely control actions, and generating alarms under critical situations. The proposed solution helps to improve pond management, reduce dependence on manual supervision, and strengthen UNCA's capacity for teaching, applied research, and aquaculture production.

Keywords: *Control and Industrial Automation, Aquaculture, Embedded systems, Water quality, SDG 2 - Zero Hunger, SDG 12 - Responsible Consumption and Production.*

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Introducción

La piscicultura constituye una actividad estratégica para la producción de alimentos y el fortalecimiento de sistemas agroproductivos sostenibles, especialmente en regiones donde los peces representan una fuente fundamental de proteína y desarrollo económico. La adecuada gestión de las condiciones ambientales dentro de los estanques es un factor determinante para garantizar la supervivencia, el crecimiento y la productividad de las especies cultivadas. Entre los parámetros más relevantes se encuentran el oxígeno disuelto, el pH, la temperatura, el dióxido de carbono, la turbidez y las concentraciones de compuestos nitrogenados, los cuales influyen directamente sobre la fisiología de los peces y la estabilidad del ecosistema acuático [1]. Desbalances en estas variables pueden generar estrés, disminución de la tasa de crecimiento, aparición de enfermedades y, en casos severos, mortalidad masiva [2], [3], [4].

La literatura especializada destaca que, para cultivos de especies como tilapia y pacú, es esencial mantener niveles de oxígeno disuelto superiores a 5 mg/L y un pH dentro del rango de 6,5 a 7,5, valores que favorecen una adecuada actividad metabólica y reducen el riesgo de condiciones adversas [1], [5]. Sin embargo, la variabilidad natural de los sistemas acuáticos, sumada a procesos como la respiración nocturna del fitoplancton, la descomposición de materia orgánica o el consumo metabólico de los peces, puede provocar fluctuaciones abruptas que requieren monitoreo continuo para su detección oportuna.

En la Facultad de Ciencias de la Producción de la Universidad Nacional del Caaguazú (UNCA), los estanques destinados a la cría de tilapia y pacú constituyen un espacio académico y productivo donde se desarrollan prácticas de enseñanza, trabajos de investigación y actividades de manejo acuícola. No obstante, el monitoreo de parámetros críticos se realizaba de forma manual, con mediciones aisladas cada dos a tres días. El oxígeno disuelto se evaluaba mediante la observación del comportamiento de los peces, lo cual resultaba subjetivo y reactivo, mientras que el pH se medía mediante tiras reactivas, un método de baja precisión y alta variabilidad. Este enfoque dificultaba la detección temprana de condiciones adversas, especialmente en horarios críticos, y derivaba en fluctuaciones bruscas del pH y del oxígeno disuelto, asociadas a estrés, enfermedades y pérdidas anuales estimadas en aproximadamente 449,2 kilogramos de peces [5].

Además, otros parámetros relevantes como el dióxido de carbono, el amonio, la turbidez o la dureza del agua no se medían, principalmente por la ausencia de instrumentos específicos y la falta de continuidad en el control. Estudios científicos resaltan que niveles elevados de CO_2 reducen el

crecimiento y el apetito de los peces [3], que el amonio y los nitritos pueden resultar tóxicos incluso a bajas concentraciones [2], y que la variación térmica abrupta provoca estrés y susceptibilidad a enfermedades [4]. La falta de un monitoreo integral en los estanques de la UNCA contrastaba con las recomendaciones descritas en literatura especializada en acuicultura, donde se enfatiza la importancia de la medición continua de variables ambientales para optimizar el rendimiento productivo y prevenir pérdidas [7].

Frente a esta problemática, surgió la necesidad de implementar una solución tecnológica que permitiera automatizar el monitoreo y la corrección de los parámetros críticos del agua, reduciendo así la dependencia de métodos manuales, mejorando la precisión de las mediciones y aumentando la eficiencia del manejo piscícola. La incorporación de sensores, sistemas embebidos, controladores PID y plataformas de supervisión en tiempo real ha demostrado ser una herramienta efectiva para estabilizar condiciones ambientales, incrementar la productividad y optimizar el uso de recursos en sistemas acuícolas [8], [9], [10].

El presente Proyecto Final de Grado se desarrolló con el objetivo de diseñar e implementar un sistema automatizado capaz de monitorear, controlar y corregir en tiempo real los parámetros críticos de calidad del agua en los estanques de piscicultura de la Facultad de Ciencias de la Producción de la UNCA. Para ello, se integraron sensores de pH, oxígeno disuelto y temperatura, actuadores de corrección y un sistema de control basado en una Raspberry Pi 4 que permite ejecutar procesos de medición continua, control automático mediante algoritmos PID, registro histórico de datos, comunicación remota y visualización de información a través de una interfaz gráfica. El desarrollo se basó en metodologías de investigación aplicada y de campo, combinando análisis cuantitativo y cualitativo para evaluar las condiciones reales del sistema y validar el desempeño del prototipo implementado.

Desde el punto de vista académico y científico, el trabajo se enmarca en la línea de investigación “Ingeniería Electrónica – Control y Automatización Industrial” de la FCyT, al aplicar sistemas embebidos y control automático para resolver una problemática real en el ámbito productivo. Asimismo, contribuye al cumplimiento del ODS 2 - Hambre cero, al favorecer una producción piscícola más estable y eficiente, y del ODS 12 - Producción y Consumo Responsables, al optimizar el uso de recursos y reducir pérdidas en el sistema de cultivo. Estos elementos justifican la pertinencia del proyecto dentro de las prioridades institucionales de la UNCA y de la Agenda 2030 para el Desarrollo Sostenible.

1. Metodología

1.1. Diseño del estudio

El presente trabajo se enmarca en una investigación aplicada de carácter tecnológico, orientada al diseño e implementación de un sistema automatizado para el monitoreo y control de parámetros críticos de calidad del agua en estanques de piscicultura. El propósito central no fue describir únicamente la situación existente, sino desarrollar, integrar y validar un prototipo funcional basado en sensores, sistemas embebidos y algoritmos de control automático, capaz de operar en condiciones reales de campo.

Se adoptó un enfoque mixto, combinando elementos cuantitativos y cualitativos.

En el componente cuantitativo, el estudio se apoyó en la medición sistemática de parámetros fisicoquímicos del agua mediante sensores electrónicos, el registro continuo de datos y el análisis de la respuesta del sistema de control frente a las variaciones observadas. Esto permitió evaluar tendencias, fluctuaciones y capacidad de corrección del prototipo a partir de evidencia objetiva.

En el componente cualitativo, se consideraron la observación directa del manejo previo de los estanques, las restricciones operativas del entorno y el intercambio con el encargado de la unidad de piscicultura, a fin de adecuar el diseño del sistema a las necesidades reales de uso y mantenimiento.

Desde el punto de vista metodológico, el estudio se clasifica como no experimental y longitudinal. No se manipularon deliberadamente las condiciones del estanque para generar escenarios controlados, sino que el sistema se evaluó bajo las variaciones naturales de los parámetros ambientales propias del cultivo. La naturaleza longitudinal se refleja en el registro continuo en el tiempo y en el seguimiento del comportamiento del pH, del oxígeno disuelto y de la temperatura durante distintos periodos de operación, tanto diurnos como nocturnos.

En términos de ingeniería, el proyecto se desarrolló bajo un diseño de prototipo que siguió una secuencia iterativa de etapas: diagnóstico de la problemática, definición de requerimientos funcionales, selección de componentes, diseño de la arquitectura hardware–software, implementación de algoritmos de control y de seguridad, integración en campo y validación del desempeño. Este enfoque permitió justificar el diseño metodológico elegido y asegurar que los procedimientos descritos puedan ser replicados o adaptados por otros investigadores o instituciones en contextos de piscicultura similares.

1.2. Área de estudio y población

El proyecto se desarrolló en la unidad de piscicultura de la Facultad de Ciencias de la Producción de la Universidad Nacional del Caaguazú (UNCA), ubicada en el kilómetro 136 de la Ruta N.º 8 “Dr. Blas Garay”. Esta unidad cuenta con dos estanques destinados al cultivo de tilapia (Oreochromis spp.) y pacú (Piaractus mesopotamicus), que constituyen un espacio de apoyo para actividades de docencia, investigación aplicada y producción acuícola institucional.

Los estanques presentan volúmenes aproximados de 300.000 L y 200.000 L de agua, respectivamente, configurando sistemas de cultivo extensivo donde las condiciones de calidad del agua (pH, oxígeno disuelto, temperatura y turbidez) influyen directamente en la supervivencia, el crecimiento y el bienestar de los peces. En este contexto, la población de estudio se corresponde con el sistema de producción acuícola de la unidad de piscicultura de la UNCA, mientras que la muestra está representada por los estanques en los que se instaló y evaluó el sistema automatizado, con énfasis en el estanque seleccionado como caso de prueba para la validación del prototipo.

Previo a la implementación del sistema, el monitoreo de los parámetros críticos se realizaba de forma manual y esporádica, a través de mediciones puntuales y de la observación del comportamiento de los peces, lo que dificultaba la detección temprana de fluctuaciones bruscas y limitaba la capacidad de intervención oportuna. El diseño metodológico del proyecto consideró estas limitaciones iniciales y orientó la solución tecnológica hacia un esquema de monitoreo continuo y corrección automática, compatible con las rutinas de manejo existentes y con los recursos disponibles en la institución.

Desde el punto de vista ético, el trabajo no implicó experimentación invasiva ni procedimientos que requirieran la manipulación directa de los peces; las intervenciones se limitaron al ajuste controlado de las condiciones físico-químicas del agua dentro de los rangos recomendados por la literatura para el cultivo de tilapia y pacú. En ese sentido, el sistema propuesto se concibió como una herramienta para mejorar el bienestar de los organismos y optimizar el manejo de los estanques, sin involucrar sujetos humanos ni recolección de datos personales.

1.3. Variables y parámetros de estudio

El sistema automatizado desarrollado se centró en el monitoreo y control de un conjunto de variables de calidad del agua y de variables operativas del propio sistema de control, seleccionadas

en función de su impacto directo sobre el bienestar de los peces y la estabilidad del proceso de cultivo.

Las variables ambientales principales consideradas fueron:

- pH del agua (adimensional): parámetro crítico para la fisiología de tilapia y pacú, utilizado además como variable de entrada para el controlador PID de corrección química.
- Oxígeno disuelto (mg/L): indicador del equilibrio entre respiración, fotosíntesis y mezcla del agua, empleado como variable controlada por el PID de oxigenación mediante aireadores.
- Temperatura del agua (°C): parámetro de referencia para interpretar las demás variables y validar la coherencia de las mediciones, así como para apoyar la compensación del sensor de oxígeno disuelto.
- Turbidez (NTU, a nivel de diseño): asociada a sólidos suspendidos y carga orgánica; fue incorporada en la arquitectura del sistema como variable de diseño para futuras ampliaciones.
- Dióxido de carbono disuelto (mg/L, estimado): calculado a partir de pH, temperatura y alcalinidad, utilizado como indicador complementario del equilibrio ácido–base del estanque.

Además, se registraron variables operativas del sistema de control, entre ellas:

- Salidas de los controladores PID para pH y oxígeno disuelto, expresadas en términos de tiempo de actuación de bombas peristálticas y aireadores dentro de cada ventana de control.
- Estados de los actuadores (bombas de pH-up y pH-down, aireadores principales) y del circuito de retardo a la desconexión basado en un temporizador 555, empleado para mantener el aireador de mezcla activo durante un intervalo controlado tras cada dosificación.
- Códigos de error y estados del sistema (control activo, en pausa, en error), empleados para evaluar la robustez del sistema de seguridad y la capacidad de detección de fallos.
- Registros de acciones manuales, tales como activaciones de actuadores desde la interfaz y confirmaciones de reanudación del sistema tras la resolución de errores.

El análisis de estas variables se orientó principalmente a verificar el correcto funcionamiento del prototipo, evaluar su capacidad para mantener el pH y el oxígeno disuelto dentro de los rangos recomendados, y comprobar la estabilidad de la operación en condiciones reales de campo, más que a la aplicación de técnicas estadísticas avanzadas.

1.4. Materiales y equipos

El sistema automatizado se construyó a partir de una plataforma hardware-software embebida, seleccionando componentes en función de su disponibilidad, precisión y compatibilidad con el entorno de cultivo acuícola.

Como unidad de procesamiento principal se utilizó una Raspberry Pi 4 Model B, encargada de ejecutar los algoritmos de control PID, la lógica de seguridad, el registro de datos y el servidor web de monitoreo. La adquisición de señales analógicas se realizó mediante una placa Arduino conectada a un convertidor ADS1115 de 16 bits, que permitió leer las salidas de los sensores de pH, oxígeno disuelto y turbidez (a nivel de diseño) con la resolución necesaria. La Raspberry Pi y el Arduino se comunicaron mediante enlace serial, y la Raspberry Pi se integró a la red local y a una VPN basada en Tailscale para permitir el acceso remoto seguro a la interfaz de supervisión.

El módulo de sensado estuvo conformado por:

- un sensor de pH con electrodo tipo BNC y módulo acondicionador GAHOU;
- un sensor de oxígeno disuelto galvánico DFRobot SEN0237, con acondicionamiento analógico y compensación por temperatura;
- un sensor digital de temperatura DS18B20 impermeable, conectado directamente a la Raspberry Pi mediante protocolo 1-Wire;
- un sensor de turbidez DFRobot Gravity, previsto en el diseño para mediciones de sólidos suspendidos;
- y un sensor DHT11 instalado dentro del gabinete para monitorear temperatura y humedad internas como medida de protección del hardware.

Para la actuación se emplearon dos bombas peristálticas NKP-DC-S10B, destinadas a la dosificación de soluciones correctivas para subir (pH-up) o bajar (pH-down) el pH del agua, así como aireadores de superficie utilizados tanto para la oxigenación del estanque como para favorecer la mezcla del reactivo dosificado. La conmutación de estos actuadores se realizó mediante módulos de relé controlados por las salidas GPIO de la Raspberry Pi.

El mezclado posterior a la dosificación se implementó mediante un circuito de retardo a la desconexión basado en un temporizador 555, gobernado por dos pines de la Raspberry Pi. Mientras la bomba peristáltica está activa, el circuito lógico mantiene encendido el relé del aireador; cuando la bomba se detiene, la Raspberry Pi envía un pulso de disparo al 555, que mantiene el aireador funcionando durante un intervalo aproximado de 30 segundos. Si el sistema vuelve a dosificar

antes de que finalice ese tiempo, otro pin acciona el reset del 555, reiniciando el temporizador. De esta forma se asegura una mezcla homogénea del reactivo químico.

El sistema se alimentó mediante una fuente de alimentación externa adecuada para los actuadores y la electrónica, complementada con un UPS específico para la Raspberry Pi, destinado a evitar apagados bruscos, preservar la integridad del sistema de archivos y permitir un cierre controlado ante cortes de energía. Todos los módulos electrónicos se instalaron en un gabinete de protección elevado respecto al nivel del agua, con paso de cables organizado y uso de canaletas o conduit donde fue necesario, a fin de minimizar riesgos por humedad, radiación solar, desplazamiento de personas o presencia de animales.

Los diagramas físicos y electrónicos del sistema se presentan en los Anexos A y B, mientras que el desglose de costos se incluye en el Anexo G.

1.5. Procedimiento de diseño e implementación del sistema

1.5.1. Arquitectura general del sistema

El sistema automatizado se diseñó con el propósito de monitorear, procesar y corregir en tiempo real los parámetros críticos de calidad del agua en los estanques de piscicultura de la Facultad de Ciencias de la Producción de la UNCA. La arquitectura general se estructuró de manera modular en capas, integrando sensores, unidades de adquisición, procesamiento, algoritmos de control, actuadores y mecanismos de registro y visualización, tal como se resume en la Figura 1.1.

En la capa de sensado, los sensores de pH, oxígeno disuelto y turbidez (a nivel de diseño) realizan la captura de los valores ambientales del estanque y se conectan a un convertidor analógico–digital de alta resolución (ADS1115) gestionado por una placa Arduino, encargada de digitalizar las señales y transmitir las lecturas crudas hacia la Raspberry Pi 4 mediante comunicación serial. La temperatura del agua se mide directamente desde la Raspberry Pi mediante un sensor DS18B20 por protocolo 1-Wire, mientras que un sensor DHT11 instalado en el interior del gabinete permite supervisar temperatura y humedad internas como parte de la protección del hardware.

En la capa de procesamiento y control, la Raspberry Pi actúa como núcleo del sistema, aplicando filtros y validaciones a las lecturas, estimando el CO_2 disuelto a partir de pH, temperatura y alcalinidad, ejecutando los controladores PID de pH y oxígeno disuelto, gestionando la lógica de seguridad y comandando los actuadores a través de salidas GPIO. Los actuadores bombas peristálticas para la corrección del pH y aireadores de superficie para la oxigenación y mezcla se

gobiernan mediante módulos de relé y un circuito de retardo a la desconexión basado en un temporizador 555, que asegura una mezcla adecuada del reactivo tras cada dosificación.

En paralelo, la Raspberry Pi se comunica con un módulo registrador independiente (logger) que almacena los datos en archivos CSV y en una base de datos SQLite, y con un servidor web Flask que provee la interfaz de usuario. Esta interfaz es accesible desde la red local y, adicionalmente, a través de una VPN Tailscale, que establece un túnel cifrado entre la Raspberry Pi y dispositivos externos autorizados. Esta integración permite visualizar el panel web y monitorear el sistema desde fuera de la red del campus sin exponer directamente los servicios a Internet, manteniendo al mismo tiempo la operación autónoma del control aun cuando no haya ningún cliente conectado.

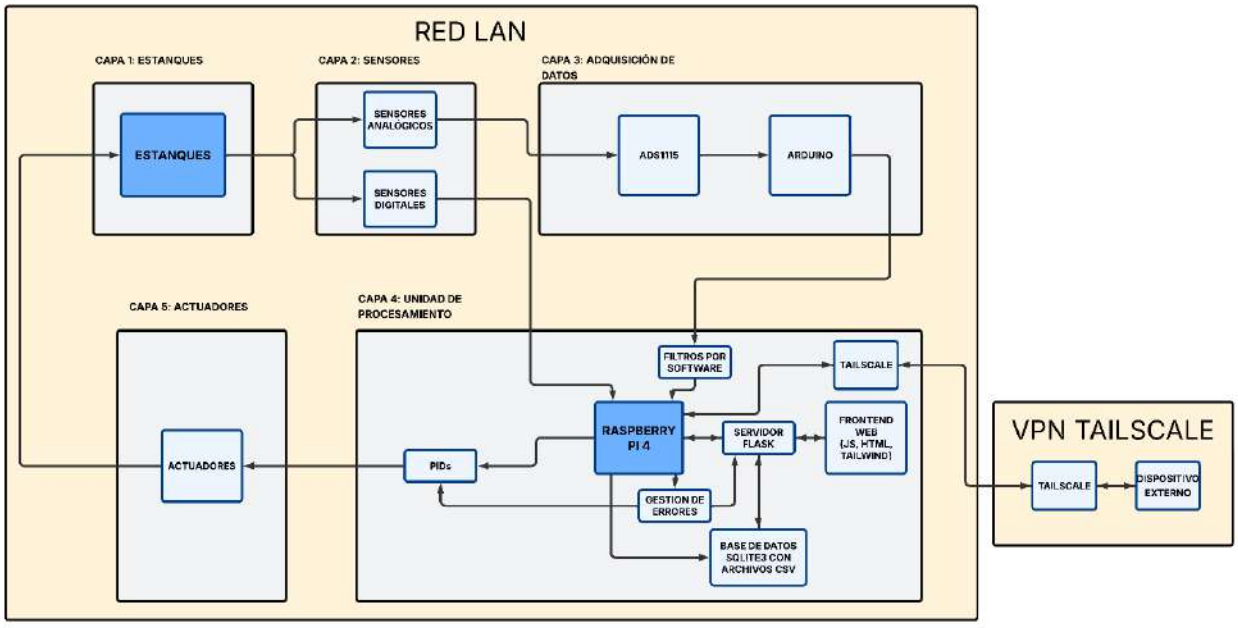

Figura 1.1 Arquitectura general del sistema automatizado

1.5.2. Módulo de medición y estimación de parámetros

El módulo de medición se diseñó para obtener lecturas continuas y estables de los parámetros críticos de calidad del agua. El pH y el oxígeno disuelto se miden con sondas analógicas conectadas a un convertidor analógico-digital ADS1115, gestionado por una placa Arduino, que transmite las lecturas digitalizadas hacia la Raspberry Pi mediante comunicación serial con un intervalo de muestreo del orden de pocos segundos. La temperatura del agua se adquiere a través de un sensor digital DS18B20 conectado directamente a la Raspberry Pi mediante el protocolo 1-Wire, mientras que el sensor de turbidez DFRobot Gravity se incorporó en la arquitectura como canal adicional de medición previsto para futuras ampliaciones del prototipo.

Sobre las lecturas recibidas, el sistema aplica un filtrado de tipo suavizado exponencial y rutinas de validación que descartan temporalmente valores fuera de rango físico, cambios abruptos incoherentes o lecturas congeladas. De esta forma se obtiene un conjunto de datos depurados y coherentes que alimenta tanto a los controladores PID como a los módulos de registro y visualización. La medición del pH constituye la referencia principal para la dosificación automática de soluciones correctivas, mientras que el oxígeno disuelto gobierna la activación de los aireadores; la temperatura se utiliza como variable de apoyo para la compensación de lecturas y la interpretación del estado del estanque.

Adicionalmente, el sistema estima el nivel de dióxido de carbono disuelto (CO_2) a partir del pH, la temperatura y un valor de alcalinidad (KH) ingresado manualmente por el usuario, utilizando una relación empírica ampliamente empleada en sistemas acuáticos. En cada ciclo de procesamiento se combinan estos valores para calcular la concentración aproximada de CO_2, que luego se registra junto con el resto de parámetros. El procedimiento de cálculo y su formulación matemática se detallan en el Anexo C.1.

1.5.3. Estrategia de control automático y seguridad

El sistema de control automático se basa en dos controladores PID independientes: uno para el pH y otro para el oxígeno disuelto (OD). Ambos operan en la Raspberry Pi sobre las lecturas filtradas de los sensores y actúan únicamente mediante actuadores de tipo ON/OFF, por lo que la salida del PID se interpreta como tiempo de actuación dentro de una ventana de control y no como una señal analógica continua.

En el caso del pH, se implementaron dos lazos PID mutuamente excluyentes: uno asociado a la bomba peristáltica de solución alcalina (pH-up) y otro a la bomba de solución ácida (pH-down). El valor de pH medido se compara con el setpoint y se calcula el error; si el valor se encuentra dentro de una banda de tolerancia (deadband), no se ejecuta ninguna corrección. Cuando el error supera dicha banda, el sistema determina la dirección de la corrección (subir o bajar el pH), habilita únicamente el PID correspondiente y convierte la salida de este en un intervalo de tiempo de dosificación. El anexo C.2 resume este flujo: desde la lectura del sensor, el cálculo del error y la selección del PID activo, hasta el mapeo de la salida a un tiempo de encendido de la bomba.

Para asegurar una mezcla homogénea del reactivo en el estanque, la activación de cualquiera de las bombas peristálticas dispara también el aireador mediante un circuito de retardo a la desconexión implementado con un temporizador 555. Mientras la bomba está encendida, la lógica

de control mantiene el aireador activo. Cuando termina la dosificación, la Raspberry Pi envía un pulso al trigger del 555, que mantiene el aireador funcionando aproximadamente 30 segundos adicionales. Si durante ese intervalo vuelve a ser necesaria una nueva corrección de pH, la placa genera un pulso sobre la entrada reset del 555 para reiniciar la cuenta. De esta forma, el aireador acompaña siempre el proceso de dosificación y la mezcla se mantiene adecuada.

El control del oxígeno disuelto sigue una lógica análoga, pero con un único lazo PID asociado a los aireadores de superficie. El nivel de OD medido se compara con el setpoint y, fuera de la banda de tolerancia, el PID calcula un tiempo de encendido dentro de cada ventana de control. Tras cada intervalo de actuación se introduce un periodo de reposo para permitir la distribución del oxígeno en el agua antes de una nueva corrección, evitando tanto la hipoxia como la sobresaturación y el consumo energético innecesario.

Sobre estos lazos de control se superpone una arquitectura de seguridad que valida cada lectura antes de permitir cualquier actuación. El sistema verifica rangos físicos plausibles, cambios abruptos entre ciclos, lecturas congeladas y errores de comunicación con el Arduino o el ADS1115. Cuando las lecturas son válidas, el ciclo continúa de forma normal; si se detecta alguna anomalía, se registra un código de error y se evalúa su prioridad. El anexo C.3 ilustra este flujo: adquisición de datos, validación, clasificación del error, selección del error prioritario y paso a un estado de pausa del PID cuando se identifica una falla crítica.

Adicionalmente, la lógica de control incorpora límites operativos que restringen la duración máxima de activación por ciclo, el número de correcciones en un intervalo dado y la cantidad total de dosificación química. Estos límites, junto con la banda muerta alrededor del setpoint, reducen el desgaste de los actuadores y previenen cambios bruscos en las condiciones del estanque. El anexo C.4 presenta de manera conceptual estos márgenes: deadband superior e inferior, tiempo mínimo entre actuaciones y límites máximos por ciclo.

En conjunto, la combinación de control PID por ventanas temporales, mezcla asistida mediante el temporizador 555 y un esquema jerárquico de validación y manejo de errores permite que el sistema automatizado corrija pH y oxígeno disuelto de forma gradual, segura y supervisable, manteniendo la estabilidad del entorno acuícola y minimizando la necesidad de intervención manual.

1.5.4. Sistema de adquisición y registro de datos

El sistema de adquisición y registro de datos permite capturar las lecturas de los sensores, distribuirlas a los módulos de control y visualización, y almacenar un historial estructurado para su análisis posterior. Para ello se combinan diferentes canales de comunicación (serial, TCP y UDP), un módulo registrador independiente y una base de datos SQLite complementada con archivos CSV diarios.

1.5.4.1. Comunicación interna entre módulos

La comunicación entre la electrónica de campo y la unidad de procesamiento se organiza de la siguiente manera:

- Arduino → Raspberry Pi (Serial).

El Arduino recibe las señales analógicas del ADS1115 (pH, oxígeno disuelto y, a nivel de diseño, turbidez) y envía líneas de texto a 115200 baudios del tipo:

valor_pH_raw,valor_OD_raw

La Raspberry Pi valida el formato, convierte los valores a unidades físicas y alimenta los algoritmos de filtrado, estimación de CO_2 y control PID.

- Raspberry Pi ↔ Interfaz de usuario (TCP).

Los comandos enviados desde la interfaz web (pausar o reanudar PIDs, activar modos manuales, confirmar salida de estado de error, etc.) se transmiten por TCP, garantizando la entrega confiable de cada instrucción y evitando la pérdida de órdenes críticas.

- Raspberry Pi → Registrador (UDP).

Las lecturas procesadas y el estado del sistema (pH, OD, temperatura, CO_2 estimado, salidas PID, códigos de error, etc.) se envían de forma continua mediante UDP a un módulo registrador (logger_3.py). Este protocolo se eligió por su baja latencia y porque no bloquea el ciclo principal de control, aceptando pérdidas ocasionales de paquetes sin comprometer el monitoreo global.

1.5.4.2. Módulo registrador de datos (logger_3.py)

El script logger_3.py funciona como un proceso independiente que:

- recibe los paquetes UDP desde el controlador;
- verifica el formato básico de los datos;

- separa cada parámetro y lo registra en archivos CSV diarios;
- actualiza un archivo de acceso rápido con la última medición;
- inserta los registros en una base de datos SQLite (monitoreo.db), configurada en modo WAL para permitir escrituras continuas mientras otros procesos realizan consultas.

De este modo se combina la persistencia histórica (CSV y exportación a XLSX) con el acceso eficiente a datos recientes desde la interfaz web. El detalle del código fuente se presenta en los anexos de programación.

1.5.4.3. Base de datos SQLite y estructura de almacenamiento

La base de datos SQLite actúa como repositorio central para mediciones, errores, notificaciones y acciones manuales. Su diseño está optimizado para consultas rápidas desde el servidor Flask, sin interferir con el ciclo de control.

A continuación, se resumen las tablas principales (Tablas 1.1 a 1.4).

1.5.4.3.1. Tabla lecturas:

Campo	Descripción
id	Identificador autoincremental.
fecha	Fecha de la medición (YYYY-MM-DD).
hora	Hora de la medición (HH:MM:SS).
ph	Valor del pH medido.
o2	Valor de oxígeno disuelto.
temp	Temperatura medida.
pid_ph_down	Salida del PID para disminuir pH.
pid_ph_up	Salida del PID para aumentar pH.
pid_o2	Salida del PID para oxigenación.
codigo_error	Código(s) de error activo(s) durante la medición.

Tabla 1.1 Variables de la tabla lecturas

1.5.4.3.2. Tabla errores_log:

Campo	Descripción
id	Identificador del evento de error.
codigo	Código numérico del error.
descripcion	Descripción textual del error.
hora_inicio	Momento en que comenzó el error.
hora_fin	Momento en que se resolvió (NULL si sigue activo).
resuelto	0 = activo, 1 = resuelto.
notificado_inicio	Si ya fue notificado el inicio del error.
notificado_fin	Si ya fue notificada su resolución.

Tabla 1.2 Variables de la tabla errores_log

1.5.4.3.3. Tabla notificaciones:

Campo	Descripción
id	Identificador de la notificación.
tipo	Tipo de mensaje (alerta, error, info).
mensaje	Contenido textual mostrado al usuario.
hora	Momento de creación de la notificación.
leida	0 = no leída, 1 = leída.

Tabla 1.3 Variables de la tabla notificaciones

1.5.4.3.4. Tabla de acciones_manual:

Campo	Descripción
id	Identificador único autoincremental correspondiente a cada acción registrada.
fecha	Fecha exacta en la que fue ejecutada la acción manual (formato YYYY-MM-DD).
hora	Hora precisa de la acción, registrada con resolución de segundos (HH:MM:SS).
accion	Código interno de la acción realizada (p. ej.: aireador_on, pid_ph_pause, parada_emergencia, etc.).
descripcion	Descripción textual generada por el sistema, indicando el evento ocurrido y el componente afectado.
valor	Parámetro adicional asociado a la acción (duración en segundos, volumen dosificado, etc.). Puede ser NULL si la acción no requiere valores adicionales.

Tabla 1.4. Variables de la tabla acciones_manual

Además de la base de datos, el sistema mantiene:

- CSV diarios, con todas las lecturas y estados del sistema.
- Archivos XLSX generados periódicamente a partir de los CSV, para análisis externo y respaldo de largo plazo.
- Un archivo fijo de acceso rápido (por ejemplo, datos.csv) que contiene solo la última medición y es utilizado por algunos módulos de visualización.

1.5.4.4. Organización temporal y sincronización

Todas las mediciones se registran con fecha y hora generadas en la Raspberry Pi, de modo que los distintos procesos (control, logger y servidor web) comparten una referencia temporal consistente. El uso de UDP no bloqueante, el modo WAL en SQLite y la separación de procesos evita que retrasos en el registro o en la interfaz afecten el ciclo de control principal.

1.5.4.5. Integración con la interfaz de usuario

La interfaz web desarrollada con Flask (app.py) y los módulos mobile.html y mobile_v2.js consultan exclusivamente la base de datos SQLite para:

- mostrar las mediciones recientes y el historial;
- visualizar el estado de los controladores y los códigos de error;
- listar notificaciones y acciones manuales.

Los comandos que el usuario ejecuta desde la web (por ejemplo, pausar o reanudar PIDs, activar modo manual, confirmar la salida de un estado de error) se envían por TCP hacia el módulo de control. La lógica del PID y la activación de los actuadores se mantienen aisladas de la interfaz gráfica: si el navegador, el dispositivo del usuario o el servidor Flask fallan, el sistema de control continúa operando de forma autónoma en la Raspberry Pi.

1.5.5. Interfaz de usuario y acceso remoto

1.5.5.1. Arquitectura de la interfaz web

La interfaz de usuario se desarrolló como una aplicación web ligera basada en Flask, HTML, JavaScript y TailwindCSS, accesible desde navegadores móviles y de escritorio conectados a la red local o a la VPN Tailscale.

El servidor Flask (app.py) actúa como intermediario entre el usuario, la base de datos SQLite (monitoreo.db) y el programa de control (v25.py). Desde Flask se sirven los archivos mobile.html y mobile_v2.js y se exponen rutas HTTP que entregan datos en formato JSON o reciben comandos del usuario.

El controlador (v25.py) mantiene una comunicación bidireccional con Flask:

- Por TCP recibe comandos externos (pausar o reanudar PIDs, activar actuadores en modo manual, confirmar reanudación tras error, etc.).
- Por UDP envía periódicamente su estado interno (lecturas actuales, estado de PIDs, cooldowns, últimas dosificaciones y códigos de error), que es almacenado por el registrador (logger_3.py) en la base de datos.

De esta manera, la lógica de control y registro permanece autónoma en la Raspberry Pi, mientras que la interfaz web se limita a consultar y visualizar información ya registrada, sin intervenir en los cálculos del PID ni en el funcionamiento básico del sistema.

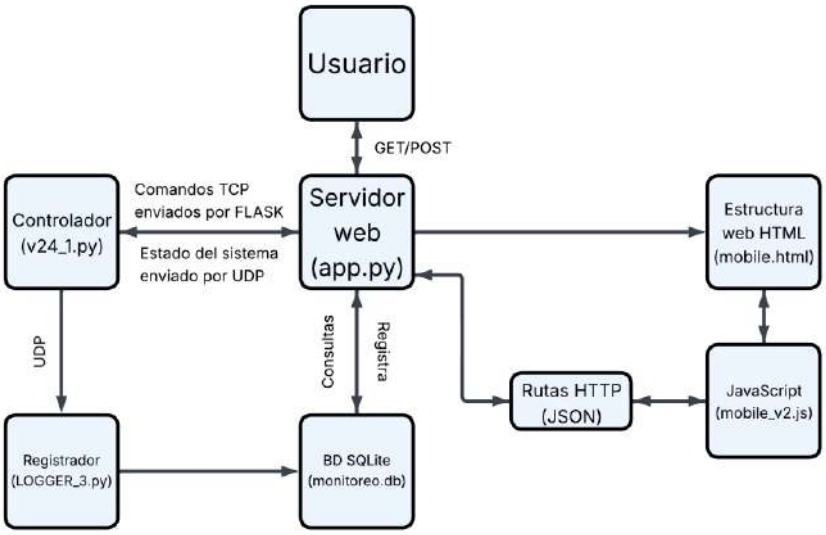

Figura 1.2. Arquitectura del sistema de visualización, registro y control

1.5.5.2. Funcionalidades principales de la interfaz

La interfaz organiza sus funcionalidades en un menú lateral optimizado para dispositivos móviles, con las siguientes secciones principales:

- Monitoreo: muestra en tarjetas los valores actuales de pH, oxígeno disuelto, temperatura del agua, turbidez y CO_2 estimado, además del estado de los PIDs y de los actuadores. También presenta información climática (temperatura ambiente, humedad y viento) obtenida desde una API externa (openWeather), la cual se consulta de forma periódica para no superar el límite de solicitudes gratuito.
- Gráficas: permite visualizar la evolución reciente de los parámetros del estanque mediante una gráfica que representa las últimas lecturas almacenadas en la base de datos, actualizadas automáticamente a través de peticiones periódicas al servidor Flask.
- Control manual: ofrece un panel para activar o desactivar el aireador, enviar dosificaciones de pH (solución alcalina o ácida), pausar o reanudar los controladores PID y ejecutar una parada de emergencia. Todos los comandos se envían al backend mediante peticiones HTTP, que Flask traduce a órdenes TCP hacia el controlador.
- Historial de lecturas y acciones: permite consultar datos históricos filtrando por tipo de registro (lecturas, acciones manuales, notificaciones), período de tiempo y estanque. Los resultados se muestran en tablas y pueden visualizarse también como gráfica para analizar tendencias.

- Notificaciones: la interfaz muestra un contador de notificaciones en un icono de campana y un panel desplegable con errores, alertas e información relevante. Cuando el usuario abre la lista, los registros se marcan como leídos en la base de datos.

La actualización de la información se realiza mediante un mecanismo de polling implementado en mobile_v2.js, que solicita cada pocos segundos un JSON con las mediciones más recientes y el estado del sistema. Esto permite un monitoreo en tiempo real sin bloquear el controlador principal.

Este esquema de navegación brinda un acceso ordenado a cada componente del sistema, manteniendo una interacción sencilla y consistente en todo momento.

En el Anexo F se incluyen capturas representativas de la interfaz web.

1.5.5.3. Seguridad y acceso (LAN/VPN Tailscale)

El acceso al panel web se limita a dispositivos conectados a la red local o a una VPN privada configurada con Tailscale.

En la red interna, el usuario ingresa al sistema mediante la dirección local de la Raspberry Pi (por ejemplo, http://192.168.x.x:5000). Para acceso remoto, Tailscale establece un túnel cifrado basado en WireGuard, asignando a la Raspberry Pi una IP virtual y un nombre de dispositivo (por ejemplo, raspberrypi), de modo que el sistema puede consultarse desde fuera del campus utilizando direcciones como:

- http://100.xxx.xxx.xxx:5000
- http://raspberrypi:5000

Este esquema evita la apertura de puertos hacia Internet, restringe el uso a dispositivos autenticados en la VPN y reduce el riesgo de accesos no autorizados, manteniendo a la vez la posibilidad de supervisar el sistema a distancia.

1.6. Calibración, pruebas y seguridad del sistema

1.6.1. Calibración y verificación de sensores

La calibración de los sensores fue una etapa clave para garantizar que los valores utilizados por el sistema de control representen de forma confiable las condiciones reales del estanque.

En el sensor de pH (electrodo BNC con módulo GAHOU) se aplicó el procedimiento estándar con soluciones tampón de pH 4,00; 7,00 y 10,00. Primero se ajustó el punto medio con el buffer de pH

7,00 y luego la pendiente utilizando los buffers ácido o básico según el rango de trabajo. Finalmente, se verificó la coherencia en los tres puntos y se comprobó que los valores convertidos por el ADS1115 coincidieran con mediciones de referencia externas.

El sensor de oxígeno disuelto SEN0237 se calibró con dos condiciones límite: agua bien oxigenada (≈100 % de saturación) y una solución con sulfito de sodio (≈0 % de OD). A partir de los voltajes medidos en ambos extremos se obtuvo una relación lineal voltaje-concentración en mg/L, incorporando la lectura del DS18B20 para compensar la dependencia del OD con la temperatura. Los valores calibrados son luego filtrados y utilizados por el PID de oxígeno.

El sensor de temperatura DS18B20 no requiere ajuste manual, pero se verificó su precisión comparándolo con un termómetro digital y uno analógico en distintos puntos de trabajo (≈20-30 °C), confirmando errores dentro del margen especificado por el fabricante (±0,5 °C).

En el caso del sensor de turbidez DFRobot Gravity, se plantea una curva de calibración que relacione el voltaje de salida con la turbidez en NTU. Como referencia se empleó la curva de respuesta del fabricante, que muestra la variación del voltaje en función de la temperatura para distintos niveles de turbidez (0, 1000 y 3000 NTU (Figura 1.3)). A partir de esta relación se definió el procedimiento para construir experimentalmente la curva voltaje-NTU en futuras ampliaciones del sistema, dejando preparada en el código la estructura necesaria para integrar dicha conversión.

El proceso de calibración de los sensores constituyó una etapa esencial para garantizar la precisión y confiabilidad de las mediciones utilizadas por el sistema automatizado. Cada sensor fue sometido a procedimientos específicos de verificación, ajuste y validación, acorde a sus principios de funcionamiento y a las recomendaciones técnicas de los fabricantes. La calibración adecuada permitió asegurar que los valores utilizados por los controladores PID reflejaran de manera fiel las condiciones reales del estanque.

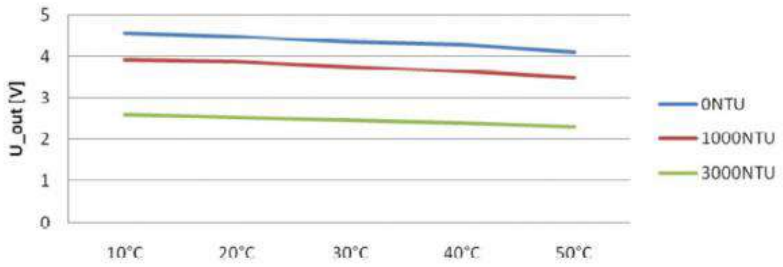

Figura 1.3. Curva de respuesta del sensor de turbidez en función de la temperatura para distintos niveles de NTU

1.6.2. Pruebas y validación del sistema

La validación del sistema se organizó en tres niveles: pruebas unitarias, pruebas funcionales e implementación en campo.

En las pruebas unitarias se verificó, de forma aislada, el correcto funcionamiento de cada componente:

- sensores (pH, OD, temperatura y turbidez),
- actuadores (bombas peristálticas, aireadores y el circuito de temporización para mezcla basado en 555, controlado desde la Raspberry Pi),
- electrónica de adquisición (ADS1115, Arduino)
- y canales de comunicación (serial Arduino-Raspberry Pi, TCP para comandos y UDP para envío de datos).

Posteriormente se realizaron pruebas funcionales del sistema integrado. Se comprobó la adquisición continua de datos en ciclos de aproximadamente 2 s, la respuesta del filtro de suavizado ante fluctuaciones artificiales y el comportamiento de los controladores PID de pH y OD frente a variaciones inducidas, evaluando estabilidad, eficacia del deadband, exclusión mutua pH-up/pH-down y activación gradual de los aireadores.

En la etapa de pruebas en los estanques de piscicultura de la Facultad, se verificó la instalación física de sensores, actuadores y gabinete, así como la continuidad del registro histórico durante periodos prolongados. Se observó la respuesta del sistema ante variaciones naturales del entorno (cambios diurnos/nocturnos en OD, fluctuaciones de pH y temperatura) y se confirmó la robustez de la comunicación y del logger bajo condiciones reales de humedad y temperatura.

Adicionalmente, se incorporó un sensor DHT11 dentro del gabinete electrónico para monitorear la temperatura y humedad interna. Se establecieron umbrales máximos y, cuando se superan, el sistema genera notificaciones al usuario, lo que permite detectar a tiempo condiciones ambientales que pudieran comprometer la electrónica.

El sistema de seguridad se evaluó mediante pruebas específicas de fallos: desconexión o simulación de lecturas congeladas de sensores, envío de cadenas corruptas por serial e interrupciones deliberadas de la comunicación Arduino-Raspberry. En todos los casos se comprobó que el sistema pase a estado de pausa, desactive actuadores, registre el error y permanezca bloqueado hasta recibir una confirmación manual. Finalmente, se verificó que los

límites operativos (tiempos máximos por ciclo, pausas mínimas y acumulados diarios) se cumplan incluso ante intentos de activación fuera de los rangos permitidos.

1.6.3. Consideraciones de seguridad y respaldo energético

El diseño incorpora varias capas de seguridad destinadas a proteger tanto a los peces como al equipamiento electrónico:

- Respaldo energético: se integró un UPS específico para la Raspberry Pi, que permite mantener el sistema activo durante cortes breves, realizar apagados controlados en interrupciones prolongadas y registrar correctamente las últimas mediciones antes del apagado, evitando corrupción del sistema de archivos o estados indeterminados de los actuadores.
- Protección eléctrica y electrónica: los módulos de control se alojan en un gabinete protegido contra humedad y polvo; se separaron físicamente las líneas de potencia (12 V de bombas y aireadores) de las líneas de señal (3,3 V y 5 V), y se utilizan relés como aislamiento entre la lógica de bajo voltaje y los actuadores. El cableado se organizó mediante canalizaciones y conduits para minimizar tensiones mecánicas y riesgos de cortocircuito.
- Seguridad química en la corrección de pH: el software impone límites máximos de tiempo por ciclo, tiempos mínimos de espera entre dosificaciones y límites diarios acumulados de reactivo. Además, la mezcla adecuada del químico se garantiza mediante un circuito temporizador basado en el integrado 555, controlado por la Raspberry Pi: mientras una bomba peristáltica está dosificando, el aireador se mantiene encendido y, al finalizar la dosificación, el temporizador prolonga su funcionamiento durante unos 30 segundos adicionales para asegurar una dispersión homogénea. El deadband evita correcciones innecesarias y, ante cualquier error crítico, las bombas quedan bloqueadas automáticamente.
- Resguardos ante fallos de sensores, comunicación o actuadores: si se detectan valores fuera de rango, lecturas congeladas, pérdida de comunicación serial o incoherencias entre el estado ordenado y el estado real de un actuador, el sistema detiene los controladores PID, apaga los actuadores y entra en modo de pausa hasta que el operador revise la situación.
- Seguridad operativa para el personal: la mayoría de las interacciones se realiza a través de la interfaz web, lo que reduce la necesidad de manipular directamente líneas de potencia o soluciones químicas. Los encendidos, apagados y pruebas de actuadores pueden ejecutarse de forma remota, disminuyendo la exposición del operador a condiciones ambientales adversas o a riesgos eléctricos.

Con este conjunto de medidas de calibración, pruebas estructuradas y mecanismos de seguridad, el prototipo alcanza un nivel de confiabilidad adecuado para su operación continua en los estanques de piscicultura de la Facultad.

1.7. Limitaciones del sistema

1.7.1. Sensibilidad y precisión de los sensores analógicos

A pesar de integrar sensado, control PID, actuación y mecanismos de seguridad, el sistema presenta limitaciones inherentes al uso de sensores analógicos, a la naturaleza ON/OFF de los actuadores y a las condiciones del entorno acuícola.

En primer lugar, los sensores de pH, oxígeno disuelto y turbidez dependen de acondicionadores analógicos y del convertidor ADS1115, por lo que sus lecturas pueden verse afectadas por ruido eléctrico, interferencias ambientales y variaciones de temperatura. El filtrado exponencial y las rutinas de validación reducen estas variaciones, pero no las eliminan completamente, por lo que se requiere una calibración periódica de los sensores de pH y OD y un mantenimiento regular de las sondas expuestas al lodo, algas y materia orgánica del estanque.

En segundo lugar, las bombas peristálticas y los aireadores son dispositivos conmutados (encendido/apagado), lo que obliga a implementar una estrategia de control basada en ventanas de tiempo en lugar de una modulación continua. Esto limita la resolución fina de las correcciones y puede generar respuestas más discretas que en un sistema con actuadores lineales.

Además, el prototipo implementado corresponde a una versión parcial del diseño completo: el sensor de turbidez y ciertas extensiones previstas para ambos estanques forman parte del alcance ampliado, por lo que algunas validaciones se realizaron en un subconjunto de condiciones. El funcionamiento continuo del sistema también depende de la estabilidad del suministro eléctrico; aunque la Raspberry Pi se protege mediante un UPS, cortes prolongados detienen la operación y requieren intervención manual.

Finalmente, la arquitectura híbrida de comunicaciones (Serial, UDP y TCP) supone una dependencia de la integridad de los enlaces. Pérdidas puntuales de paquetes UDP o fallos momentáneos en la comunicación serial pueden producir descartes de datos o retrasos en la actualización, si bien el sistema está diseñado para que estas situaciones no comprometan la seguridad de los peces ni la estabilidad general del control.

1.7.2. Procedimiento general de operación del sistema

La operación del sistema sigue una secuencia ordenada que abarca la puesta en marcha, el monitoreo y control automático, la interacción con la interfaz web, la gestión de errores y el apagado controlado.

1.7.2.1. Puesta en marcha.

Al energizar el gabinete electrónico, la Raspberry Pi ejecuta automáticamente el programa principal de control, iniciando la lectura de sensores, el filtrado y validación de datos, la ejecución de los controladores PID y la comunicación con los módulos externos. De forma paralela, el Arduino comienza a adquirir las señales analógicas a través del ADS1115 y a enviarlas por puerto serial. El UPS asociado a la Raspberry Pi estabiliza la etapa de arranque y protege frente a fluctuaciones breves del suministro.

1.7.2.2. Monitoreo continuo y control automático.

Una vez inicializado, el sistema realiza ciclos de muestreo periódicos en los que registra pH, oxígeno disuelto, temperatura, (turbidez en el diseño completo), CO_2 estimado y estado interno del controlador. Estos datos se utilizan simultáneamente para alimentar los PID de pH y OD, para el registro histórico (logger_3.py) y para la interfaz web. Mientras no exista un error crítico, los controladores:

- comparan cada lectura con su setpoint,
- aplican deadbands para evitar correcciones innecesarias,
- calculan tiempos de actuación para bombas y aireadores dentro de ventanas temporales.

Durante la corrección de pH, la bomba peristáltica correspondiente se activa y, en paralelo, el aireador se gobierna mediante un temporizador basado en el integrado 555, controlado por la Raspberry Pi, que prolonga la mezcla unos segundos después de finalizar la dosificación, asegurando una dispersión homogénea del reactivo. En condiciones normales no se requiere intervención humana.

1.7.2.3. Interacción con la interfaz de usuario.

El operador puede acceder al sistema desde un dispositivo móvil o una computadora a través de la interfaz web servida por Flask. Desde allí es posible visualizar valores en tiempo real, consultar el historial, supervisar el estado de los controladores PID, recibir notificaciones de error y enviar

comandos (pausa, reanudación, acciones manuales) a través de una comunicación TCP confiable, sin que la interfaz intervenga directamente en los cálculos del control.

1.7.2.4. Gestión del estado y tratamiento de errores.

El sistema opera en distintos estados: operación normal, zona de deadband, pausa manual y estado de error. Cuando se detecta una falla crítica en sensores, comunicación o actuadores, el sistema:

- detiene inmediatamente los controladores PID,
- desactiva todos los actuadores,
- registra el código de error y lo notifica a la interfaz,
- permanece en pausa hasta que el usuario envíe una confirmación explícita de reanudación.

Antes de reactivar el control, el software verifica que las condiciones medidas vuelvan a ser coherentes y seguras.

1.7.2.5. Finalización del funcionamiento.

Para apagar el sistema se ejecuta un procedimiento controlado sobre la Raspberry Pi, permitiendo cerrar correctamente archivos CSV, bases de datos y servicios en ejecución. El UPS asegura el tiempo necesario para este apagado seguro, tras lo cual se desenergizan los actuadores y se procede, si es necesario, al retiro de sondas para mantenimiento.

1.7.3. Síntesis del flujo metodológico

El desarrollo del sistema siguió un flujo metodológico estructurado que combinó diagnóstico del problema, diseño del prototipo, implementación tecnológica y validación en campo.

En una primera etapa se realizó el diagnóstico de la situación de los estanques de la Facultad, identificando las limitaciones del monitoreo manual y la necesidad de automatizar el control de pH y oxígeno disuelto. Sobre esta base se definió el diseño conceptual, seleccionando sensores, actuadores, electrónica de adquisición (Arduino + ADS1115), unidad de procesamiento central (Raspberry Pi) y la plataforma de visualización web.

Posteriormente se desarrolló el sistema de medición, integrando conversión analógico–digital, filtrado, validación de datos y estimación de CO_2, y se implementaron los algoritmos de control PID con deadbands, exclusión mutua en la corrección de pH, límites de seguridad y control por ventanas temporales. En paralelo se integró el sistema de actuación (bombas, aireadores y

temporizador 555 para mezcla), la arquitectura de comunicación interna y registro de datos (Serial, UDP, TCP; CSV + SQLite) y la interfaz de usuario accesible por LAN/VPN.

Finalmente, el prototipo fue sometido a pruebas unitarias, pruebas funcionales integradas y ensayos en los estanques reales, incluyendo simulaciones de fallas para validar el sistema de seguridad. La operación continua del sistema permitió comprobar su capacidad para monitorear parámetros críticos, aplicar correcciones de forma autónoma y segura, registrar información histórica y mantener condiciones adecuadas para el cultivo de peces en el contexto de la piscicultura de la UNCA.

2. RESULTADOS Y ANÁLISIS

2.1. Resultados generales de la implementación en campo

Las pruebas del sistema automatizado se desarrollaron a partir de las primeras fases de implementación del prototipo y continuaron durante un periodo prolongado, con el objetivo de evaluar su desempeño en condiciones reales de operación dentro del estanque de pacú de 300.000 litros perteneciente a la Facultad de Ciencias de la Producción de la UNCA. Durante este proceso, el sistema operó de manera continua, registrando datos en intervalos regulares y ejecutando los algoritmos de control correspondientes.

El conjunto de pruebas incluyó tanto observaciones vinculadas al comportamiento natural del estanque como intervenciones controladas destinadas a someter al sistema a condiciones atípicas, con el fin de validar la estabilidad del controlador, la robustez de la lógica de seguridad y la respuesta de los actuadores.

Los resultados obtenidos se presentan a continuación, estructurados según los parámetros críticos monitoreados y las acciones del sistema.

2.1.1. Comportamiento del pH

Durante los ensayos prolongados, el pH del estanque se mantuvo dentro de un rango estrecho, con valores típicamente comprendidos entre 7,2 y 7,4. Las series de datos registradas muestran una evolución suave, sin oscilaciones abruptas ni caídas súbitas.

En condiciones normales de operación, el controlador PID de pH permaneció inactivo durante la mayor parte del tiempo, debido a que la variable se mantuvo dentro del margen establecido como zona muerta (deadband). En las pruebas de validación se introdujeron perturbaciones controladas en el valor de pH simulado, forzando desviaciones por encima del rango de tolerancia. En estos casos, el sistema respondió activando la bomba correspondiente al modo de corrección descendente, generando salidas PID proporcionales al error inducido y dosificaciones acotadas dentro de los límites programados.

2.1.2. Comportamiento del oxígeno disuelto

Las mediciones de oxígeno disuelto mostraron valores coherentes con el comportamiento habitual del estanque, ubicándose mayoritariamente en torno a 5,0–5,5 mg/L. Durante la mayor parte del

periodo evaluado, el controlador PID asociado permaneció sin acción, debido a que las mediciones se mantenían dentro del deadband configurado. Al igual que en el caso del pH, se aplicaron variaciones forzadas a la señal de entrada del sensor, simulando una caída abrupta del oxígeno disuelto. Ante esta situación inducida, el controlador detectó la desviación y generó una salida positiva, activando el aireador durante el tiempo correspondiente. Este comportamiento se reflejó tanto en las salidas PID como en los registros de tiempo de activación del actuador.

El sistema demostró capacidad para distinguir entre fluctuaciones normales y situaciones de riesgo, así como para ejecutar la respuesta adecuada bajo condiciones de ensayo controladas.

2.1.3. Comportamiento de la temperatura

La temperatura del estanque mostró una evolución gradual a lo largo del periodo de prueba, iniciando en torno a 27 °C durante las primeras horas y aumentando progresivamente hasta valores próximos a 31-32 °C conforme avanzó la jornada. Esta tendencia es característica de los cuerpos de agua poco profundos expuestos a radiación solar directa.

El sensor DS18B20 mantuvo estabilidad en todas sus lecturas, sin picos anómalos ni interrupciones, lo que confirma la adecuada instalación y protección del módulo de medición térmica.

2.1.4. Desempeño de los controladores PID

2.1.4.1. Control del pH

El PID de pH registró salidas distintas de cero únicamente durante los periodos en que se introdujeron perturbaciones controladas en la señal de entrada. En operación normal, la salida se mantuvo en cero, reflejando un funcionamiento consistente con la lógica de zona muerta y la estabilidad del estanque durante el periodo de prueba. Las acciones correctivas que se registraron se mantuvieron dentro de los límites temporales fijados en el diseño y se tradujeron en dosificaciones discretas de corta duración.

2.1.4.2. Control del oxígeno disuelto

El PID de oxígeno disuelto mostró un comportamiento similar: la mayor parte del tiempo su salida fue nula y solo se observaron activaciones al simular condiciones de baja concentración de OD. Los tiempos de encendido de los aireadores determinados por el controlador se mantuvieron acotados, sin activaciones prolongadas ni ciclos excesivamente frecuentes.

2.1.5. Activación de actuadores

Los registros históricos indican que las bombas peristálticas y los aireadores se activaron únicamente cuando el PID asociado generó una salida distinta de cero y no existían errores activos en el sistema.

En todos los casos, los tiempos de encendido se ajustaron a los límites máximos establecidos por ciclo, y en el control de pH se respetó en todo momento la exclusión mutua entre las bombas de corrección ascendente y descendente. La transición a estados seguros ante la presencia de errores (fallos de sensores, problemas de comunicación) se reflejó en la desactivación inmediata de los actuadores y en la ausencia de nuevas órdenes hasta la confirmación manual de reanudación.

2.1.6. Comportamiento del sistema ante perturbaciones inducidas

Como parte del proceso de validación, se realizaron pruebas con perturbaciones artificiales sobre las señales de entrada, introduciendo valores extremos o físicamente improbables para pH y oxígeno disuelto, así como simulaciones de fallos en la comunicación serial y en los sensores.

Frente a estas condiciones, el sistema mostró:

- detección rápida de lecturas incoherentes,
- generación de códigos de error correspondientes,
- activación de la lógica de pausa de los controladores,
- desactivación de los actuadores,
- registro de los eventos en la base de datos y en los archivos de log.

En los escenarios en que la perturbación se interpretó como una desviación corregible (p.ej. cambio brusco de pH dentro del rango de operación), el sistema respondió con acciones de control proporcionales, mientras que, ante fallos críticos, se priorizó la seguridad deteniendo completamente el control automático.

2.2. Conclusión general del capítulo

Los resultados obtenidos en la implementación en campo muestran que el sistema automatizado opera de manera estable y coherente con el diseño propuesto. Las variables de pH y oxígeno disuelto se mantuvieron, durante el periodo de evaluación, dentro de rangos compatibles con la literatura especializada para cultivos de peces de agua dulce, lo que indica que el estanque se mantuvo en condiciones adecuadas para el desarrollo de las especies objetivo.

Las intervenciones controladas evidenciaron que los controladores PID son capaces de generar acciones correctivas proporcionales ante desviaciones significativas, respetando los límites de actuación establecidos y evitando respuestas excesivas. La combinación de control por ventanas temporales, zonas muertas y límites de tiempo por ciclo permitió asegurar correcciones graduales, acordes con la dinámica de un volumen de agua de 300.000 litros.

El sistema de adquisición y registro funcionó de forma continua, sin pérdida apreciable de información, y permitió reconstruir la evolución temporal de las variables y de las salidas de control. La integración con la base de datos SQLite y la estructura de archivos CSV y XLSX garantiza la trazabilidad de los datos y facilita su análisis posterior, tanto para la evaluación técnica del prototipo como para la toma de decisiones en la gestión del estanque.

La arquitectura de seguridad, basada en códigos de error jerarquizados, pausas automáticas del control y bloqueo de actuadores en presencia de fallos, se comportó de acuerdo con lo previsto, priorizando en todo momento la protección del ecosistema y del equipamiento electrónico.

Finalmente, los planos físicos y electrónicos del sistema (Anexos A y B), la estructura del proyecto de software (Anexo C) y el presupuesto detallado de implementación (Anexo G) complementan los resultados presentados en este capítulo, mostrando que la solución propuesta es técnicamente viable, reproducible y escalable para su aplicación en otros sistemas acuícolas con características similares.

3. Conclusiones Y Recomendaciones

3.1. Conclusiones

- El sistema automatizado desarrollado integró correctamente los sensores de pH, oxígeno disuelto y temperatura, junto con la estimación de CO_2, proporcionando mediciones continuas y estables adecuadas para el monitoreo del estanque de 300.000 litros.
- Los controladores PID implementados para pH y oxígeno disuelto demostraron un comportamiento estable y proporcional, aplicando correcciones graduales mediante control por ventanas temporales, zonas muertas (deadbands) y límites de actuación definidos en el diseño.
- La estrategia de actuación basada en bombas peristálticas y aireadores, combinada con la lógica de mezcla asistida por el aireador controlado mediante un temporizador de retardo a la desconexión con circuito 555, permitió aplicar las correcciones químicas sin generar cambios bruscos en el ecosistema acuícola.
- La arquitectura de comunicación basada en enlaces seriales, UDP y TCP permitió separar de forma robusta las funciones de control, registro e interfaz de usuario, manteniendo la operación autónoma del sistema aun cuando la interfaz web no se encontraba disponible.
- El módulo registrador y la base de datos SQLite garantizaron la trazabilidad de la operación del sistema, facilitando el análisis posterior del comportamiento de las variables medidas, de las salidas PID y de la activación de los actuadores.
- El sistema de seguridad, basado en detección y jerarquización de errores con pausas automáticas del control y bloqueo de actuadores, resultó eficaz para prevenir actuaciones peligrosas ante fallos de sensores, interrupciones de comunicación o inconsistencias en la respuesta de los actuadores.
- La interfaz web desarrollada permitió un monitoreo local y remoto claro y accesible, la consulta de históricos y el envío de comandos de control manual, sin interferir con la estabilidad del lazo de control automático implementado en la Raspberry Pi.
- Aunque algunas funcionalidades previstas en el diseño completo como la medición efectiva de turbidez y la integración simultánea de ambos estanques no fueron implementadas en esta versión, el prototipo demostró ser una solución técnicamente viable, modular y escalable para su aplicación en sistemas acuícolas similares.

3.2. Recomendaciones

- Implementar la medición y calibración completa de la turbidez del agua, incorporando la curva voltaje–NTU en el código y su visualización en la interfaz web, de manera a enriquecer el diagnóstico de la calidad del agua.
- Extender la arquitectura desarrollada al segundo estanque de la Facultad, evaluando el desempeño del sistema en un escenario multiestanque y ajustando, si es necesario, la estructura de comunicaciones y de registro de datos.
- Incorporar herramientas seguras para ajustar setpoints y parámetros PID desde la interfaz de usuario, con restricciones de acceso, a fin de facilitar la sintonización en campo sin modificar directamente el código fuente.
- Establecer y documentar un plan de mantenimiento preventivo que incluya limpieza y revisión periódica de sondas, verificación de calibración de sensores, control del estado de las bombas peristálticas y comprobación del circuito de aireación.
- Evaluar la ampliación del sistema de respaldo energético, aumentando la autonomía del UPS e incorporando protecciones adicionales, especialmente en contextos con alta frecuencia de cortes de energía.
- Desarrollar módulos adicionales de análisis de datos e indicadores en la interfaz web, tales como tendencias por día o semana, alarmas tempranas y reportes exportables, para apoyar la toma de decisiones en la gestión del cultivo.
- Utilizar el presupuesto detallado presentado en el Anexo E como base para elaborar propuestas técnico–económicas adaptadas a futuras implementaciones en otras unidades de producción acuícola, considerando tanto la inversión inicial como los costos de operación y mantenimiento.

4. Bibliografía

[1] D. E. D. Balbuena Rivarola, Manual Básico de Piscicultura para Paraguay, 2011.

[2] J. R. Tomasso, «Toxicity of nitrogenous wastes to aquaculture animals,» *Taylor & Francis,* 1994.

[3] P. V. Skov, 8 - CO2 in aquaculture, 2019.

[4] M. Mugwanya, M. A.O. Dawood, F. Kimera y H. Sewilam, «Anthropogenic temperature fluctuations and their effect on aquaculture: A comprehensive review,» *ScienceDirect,* vol. 7, 2022.

[5] C. E. Piña Lopez , PISCICULTURA, Bogotá, 1992.

[6] D. L. Kramer, «Dissolved Oxygen and fish behavior,» *Environmental Biology of Fishes,* pp. 81-92, 1987.

[7] P. C. S. John S. Lucas, Aquaculture: Farming Aquatic Animals and Plants, Blackwell Publishing Ltd., 2012.

[8] J. P. Gonzalez Lugo, D. A. Montiel Lugo y J. Gómez Gómez, «Desarrollo de un sistema de adquisición de parámetros ambientales en los estanques piscícolas basado en IoT,» *Dialnet,* vol. 4, nº 2, p. 10, 2021.

[9] D. A. H. F. J. &. C. S. R. Skoog, Principios de análisis instrumental, Cengage Learning Editores, 2008.

[10] Kalstein France, «Kalstein,» Kalstein France (fabricante del contenido y de los equipos), 2022. [En línea].

5. Anexos

Anexo A. Diagrama y planos físicos del sistema

Anexo A.1. Diagrama general del sistema

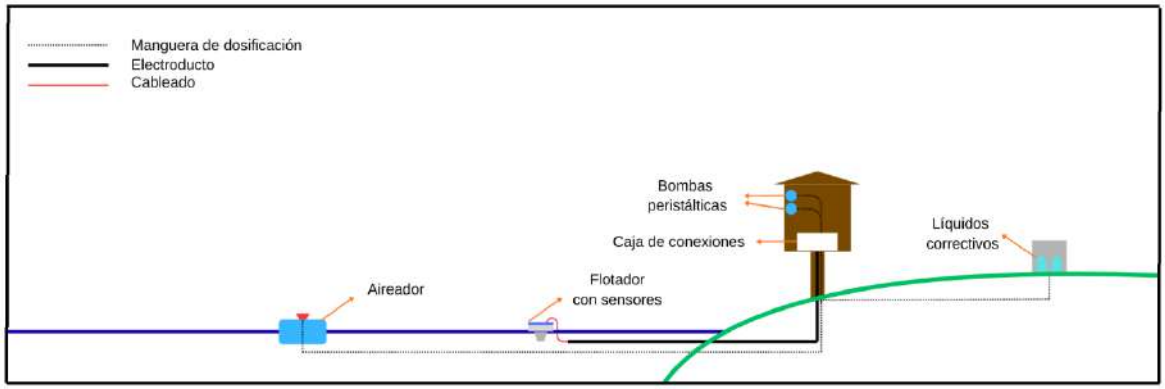

Figura 5.1 Representación esquemática del sistema instalado en el estanque

Anexo B. Diagrama electrónico

Anexo B.1. Diagrama esquemático general del sistema de monitoreo y control

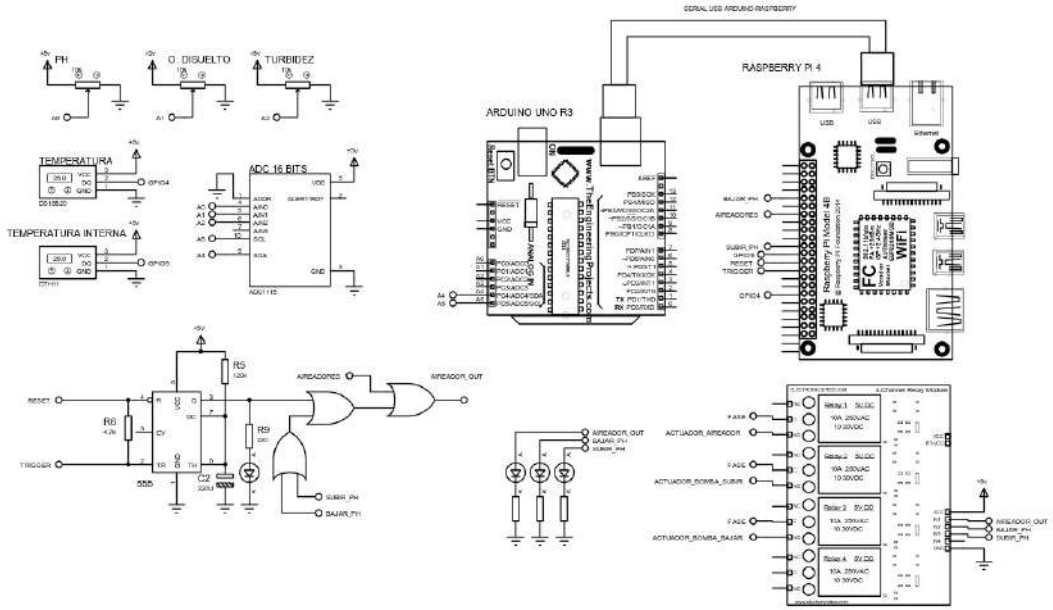

Figura 5.2 Diagrama esquemático completo del sistema de monitoreo del estanque

Anexo C. Diagramas de flujo y cálculos complementarios

Anexo C.1. Proceso de estimación del CO2 disuelto

$$CO_2\left(\frac{mg}{L}\right) \approx 3 \cdot KH\ (^\circ dKH) \cdot 10^{(7-pH)}$$

Ecuación 1 Estimación del CO_2 disuelto a partir de pH y KH.

Figura 5.3. Proceso de estimación del CO_2 a partir de pH y KH

Anexo C.2. Estrategia de control PID para el pH

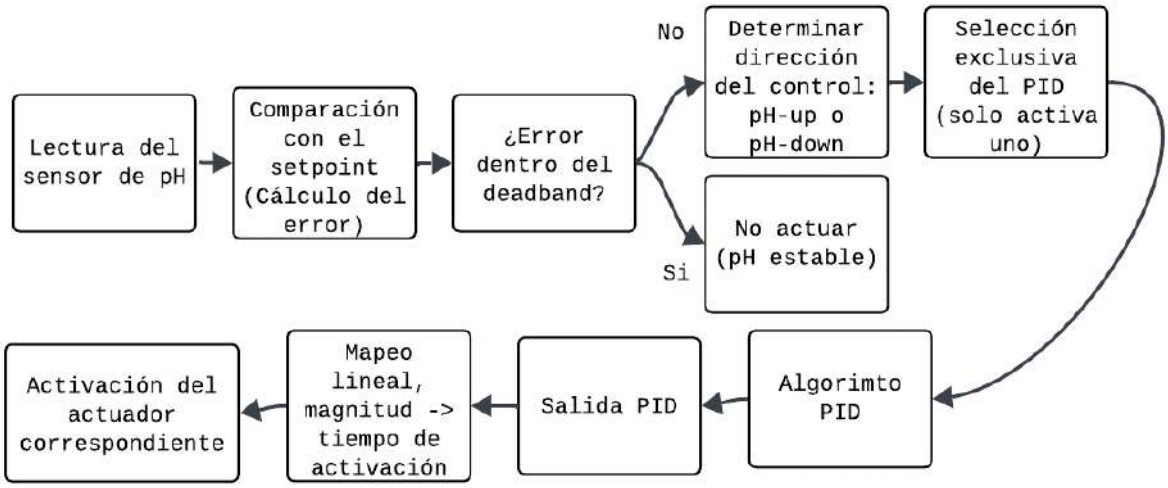

Figura 5.4. Flujo del control PID aplicado al pH

Anexo C.3. Flujo de validación y gestión de errores

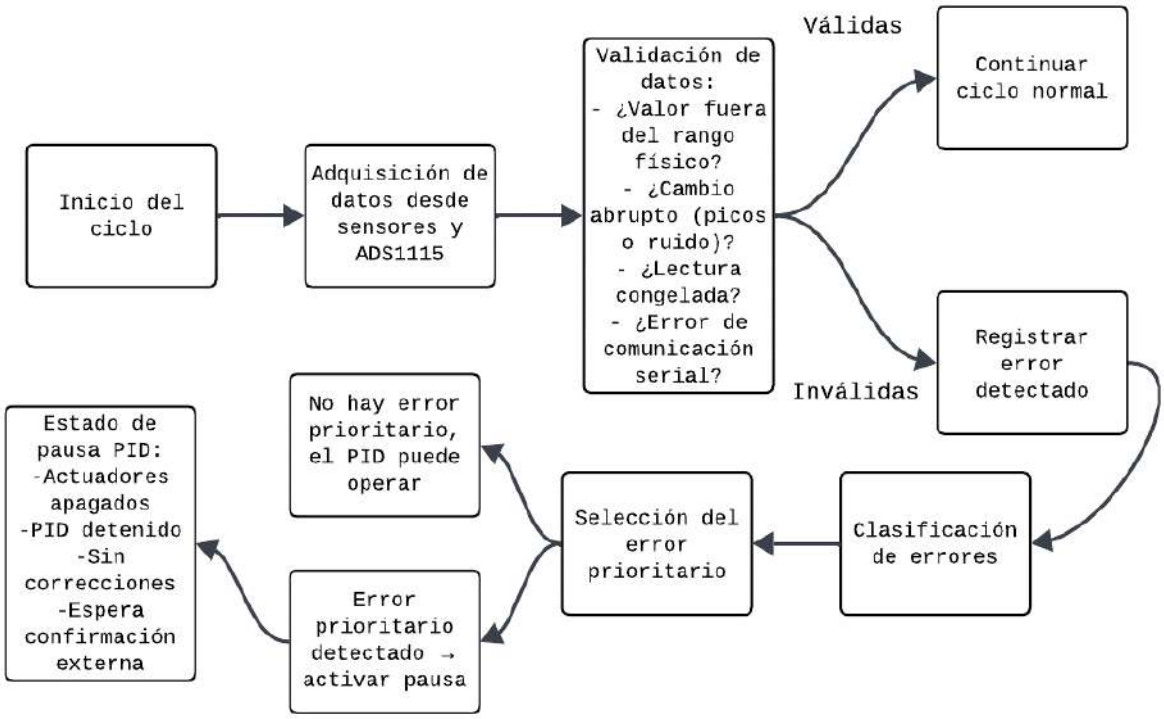

Figura 5.5. Flujo del sistema de validación y gestión de errores

Anexo C.4. Límites operativos del control

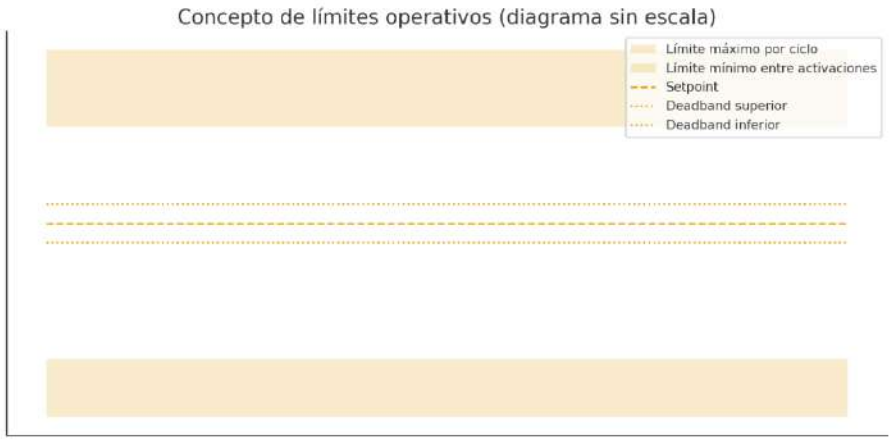

Figura 5.6. Representación conceptual de los límites operativos del control

Anexo D. Estructura del proyecto de software

Anexo D.1. Estructura de archivos

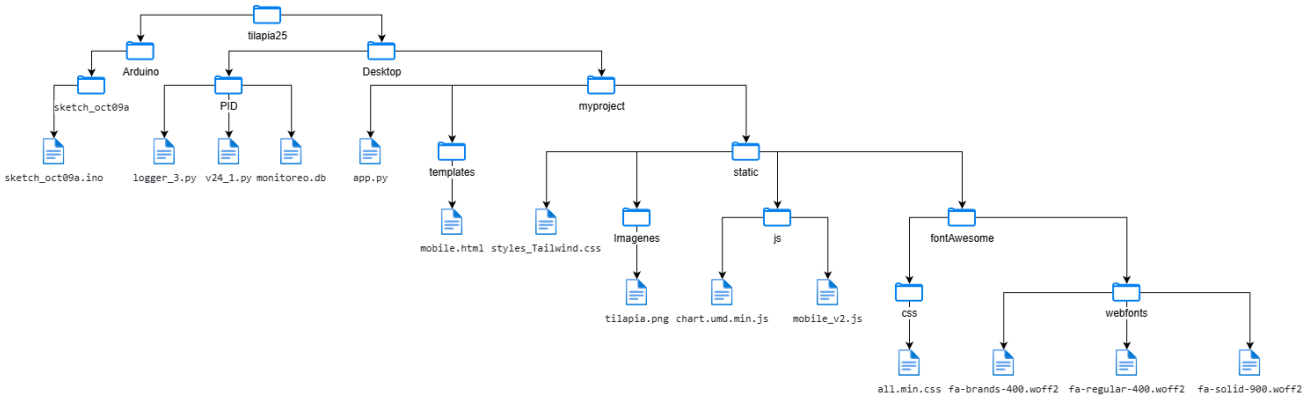

Figura 5.7 Diagrama visual de la estructura de archivos del proyecto

Anexo D.1.1. Descripción de archivos del proyecto

Archivo / Carpeta	Ubicación	Descripción de la función
tilapia25/	Carpeta raíz	Contiene todos los archivos del proyecto.
Arduino/	tilapia25/Arduino	Carpeta que contiene el código del microcontrolador Arduino.
sketch_oct09a.ino	Arduino/sketch_oct09a	Programa del Arduino. Su función es recibir las señales del ADS1115 (sensores analógicos), leerlas y transmitirlas vía Serial a la Raspberry Pi.
PID/	tilapia25/PID	Contiene los módulos Python asociados al control, registro y lógica del sistema.
logger_3.py	PID/logger_3.py	Script encargado de recibir datos vía UDP y almacenarlos en archivos CSV y en la base de datos.
v24_1.py	PID/v24_1.py	Implementa lógica de control manual, pausa de PID, comunicación con el servidor web por UDP/TCP.
monitoreo.db	PID/monitoreo.db	Base de datos SQLite utilizada para almacenar lecturas históricas y notificaciones del sistema.
Desktop/myproject/	tilapia25/Desktop/myproject	Carpeta principal del proyecto web y backend en Raspberry.
app.py	myproject/app.py	Servidor Flask. Expone API REST para la interfaz web y gestiona solicitudes del usuario.

templates/mobile.html	myproject/templates	Archivo HTML principal de la interfaz gráfica web/móvil.
static/	myproject/static	Contiene recursos estáticos para la interfaz: imágenes, CSS, JS y fuentes.
styles_Tailwind.css	static/styles_Tailwind.css	Hoja de estilos principal del frontend, generada con Tailwind.
Imagenes/tilapia.png	static/Imagenes	Imagen utilizada como ícono del sitio web
js/mobile_v2.js	static/js/mobile_v2.js	Lógica completa de la interfaz: menús, botones, gráfica, modal de errores, comunicación con API.
js/chart.umd.min.js	static/js/chart.umd.min.js	Biblioteca Chart.js usada para las gráficas en tiempo real.
fontAwesome/	static/fontAwesome	Íconos y recursos visuales (CSS + fuentes).
all.min.css	static/fontAwesome/css	Hoja de estilos de FontAwesome, para iconografía usada en botones y menús.
fa-brands-400.woff2, fa-regular-400.woff2, fa-solid-900.woff2	static/fontAwesome/webfonts	Fuentes utilizadas por FontAwesome.

Tabla 5.1 Descripción de la estructura de archivos utilizada en el sistema de monitoreo y control

Anexo E. Programación del sistema

El presente anexo reúne los fragmentos de código más relevantes del sistema automatizado (Arduino, controlador principal en Raspberry Pi, módulo registrador y servidor Flask). Para una consulta exhaustiva del código fuente, incluyendo los archivos de la interfaz web mobile.html y mobile_v2.js, se dispone de un repositorio público en GitHub

Repositorio disponible en: **https://github.com/Fredy-Lopez/PFG-ESTANQUES-FCP**

Anexo E.1. Programación del Arduino

Anexo E.1.1. Código de sketch_oct09a.ino

```
#include <Wire.h>
#include <Adafruit_ADS1X15.h>

// Crea el objeto para el ADS1115. Dirección por defecto 0x48.
Adafruit_ADS1115 ads;

void setup(void)
{
  // Iniciamos la comunicación serial a 115200 baudios.
```

```
  Serial.begin(115200);

  // No imprimimos mensajes de inicio o error. Si hay un problema,
  // el código simplemente se detendrá en la inicialización sin enviar nada.

  // Esperamos un momento para que el chip esté listo
  delay(100);

  // Inicializa el ADS1115 y lo configura para FSR de +/-6.144V
  if (ads.begin()) {
    ads.setGain(GAIN_TWOTHIRDS); // Configura GAIN_TWOTHIRDS (+/-6.144V)
  }
  // Si ads.begin() falla, no hacemos nada y no enviamos nada al serial.
}

void loop(void)
{
  int16_t adc0_raw;
  int16_t adc1_raw;

  // 1. Lectura del valor RAW del canal A0 (Single-Ended)
  adc0_raw = ads.readADC_SingleEnded(0);

  // 2. Lectura del valor RAW del canal A1 (Single-Ended)
  adc1_raw = ads.readADC_SingleEnded(1);

  // 3. Envío de datos al Monitor Serial (separados por coma)
  // Formato estricto: RAW_A0,RAW_A1\n

  Serial.print(adc0_raw);
  Serial.print(",");
  // Usamos println para añadir el salto de línea y finalizar el registro
  Serial.println(adc1_raw);

  // Pequeño retardo para controlar la tasa de muestreo
  delay(100);
}
```

Anexo E.2. Programación de la raspberry pi 4

Anexo E.2.1. Código de v25.py

```
# v25

# ================================================
# CAMBIOS Y MEJORAS v25
# ================================================
#
# ================================================

import serial
import time
from simple_pid import PID
import RPi.GPIO as GPIO
from w1thermsensor import W1ThermSensor
import socket
import statistics
```

```
import subprocess
import os
import sys
import json
from datetime import datetime
import requests

errores_activos = set()     # conjunto de códigos de error activos
codigo_error_actual = 0     # último código de error enviado o registrado

# --- NUEVO: Estados de pausa manual de PIDs ---
pid_paused_ph = False       # True = PID pH detenido hasta confirmacion externa
pid_paused_o2 = False       # True = PID O2 detenido hasta confirmacion externa

# --- PRIORIDAD DE ERRORES ---
PRIORIDAD_ERRORES = {       # entre más bajo el número, mayor prioridad (error ,prioridad)
    0: 99,                  # sin error
    13: 1,                  # logger
    14: 1,                  # apertura de puerto serial
    15: 2,                  # lectura / reconexión serial
    16: 2,                  # error genérico serial
    17: 3,                  # DS18B20
    18: 4,                  # UDP
    19: 5,                  # GPIO
    20: 6,                  # bucle principal
    1: 10, 2: 10, 3: 10,
    4: 11, 5: 11, 6: 11,
    7: 12, 8: 12, 9: 12,
    10: 13, 11: 14, 12: 14
}

# ==============================
# INICIO AUTOMÁTICO DEL REGISTRADOR CSV
# ==============================
def iniciar_logger_csv():
    """Inicia el script logger_3.py en segundo plano."""
    ruta_logger = os.path.join(os.path.dirname(os.path.abspath(__file__)), "logger_3.py")
    try:
        subprocess.Popen(               # Lanza el proceso en segundo plano sin bloquear el script
principal
            ["python3", ruta_logger],
            stdout=subprocess.DEVNULL,  # suprime salida estándar
            stderr=subprocess.DEVNULL   # suprime mensajes de error
        )
    except Exception as e:
        errores_activos.add(13)
        pass                            # Si falla el lanzamiento, el sistema principal continúa

iniciar_logger_csv()                    # Ejecutar el logger automáticamente al iniciar el sistema
principal
time.sleep(1.0)                         # breve espera para que el logger se inicialice

# ==============================
sensor_ds18b20 = W1ThermSensor()        # Inicializa sensor de temperatura DS18B20
# ==============================
# CONFIGURACION SERIAL
# ==============================
```

```
SERIAL_PORT = '/dev/ttyACM0'
BAUD_RATE = 115200

N_MUESTRAS = 5             # Número de muestras para promedio
CONVERSION_FACTOR = {
    'A0': 0.0001875,       # ±6.144 V (sensor 0–5 V)
    'A1': 0.000125         # ±4.096 V (sensor 0–3 V)
}
CANALES = ['A0', 'A1']
ALPHA_SUAVIZADO = 0.8      # Factor de suavizado exponencial
DELAY_MUESTREO = 0.05      # segundos entre iteraciones del bucle

# Lee temperatura cada X segundos (para no bloquear el bucle en cada iteración)
TEMP_READ_INTERVAL = 5.0   # segundos

# --- Estado interno de cache de temperatura ---
_last_temp_read = 0.0
_temp_cache = 25.0

# ==============================
# CONFIGURACION PID
# ==============================
Kp_o2, Ki_o2, Kd_o2 = 2.0, 0.5, 0.1                    # Ganancias O2
Kp_ph_up, Ki_ph_up, Kd_ph_up = 1.5, 0.5, 0.7           # Ganancias pH Up
Kp_ph_down, Ki_ph_down, Kd_ph_down = -2.0, -0.5, -0.7  # Ganancias pH Down

# --- Setpoints ---
setpoint_o2 = 5.0  # mg/L
SETPOINT_pH = 7.5
setpoint_ph_up = SETPOINT_pH
setpoint_ph_down = SETPOINT_pH

# --- Banda muerta ---
DEADBAND_O2 = 0.5  # mg/L
DEADBAND_PH = 0.4  # pH

# --- Objetos PID ---
pid_o2 = PID(Kp_o2, Ki_o2, Kd_o2, setpoint=setpoint_o2)
pid_ph_up = PID(Kp_ph_up, Ki_ph_up, Kd_ph_up, setpoint=setpoint_ph_up)
pid_ph_down = PID(Kp_ph_down, Ki_ph_down, Kd_ph_down, setpoint=setpoint_ph_down)

# --- Límites de salida ---
pid_o2.output_limits = (0, 100)
pid_ph_up.output_limits = (0, 100)
pid_ph_down.output_limits = (0, 100)

# ==============================
# CONFIGURACIÓN UDP/TCP
# ==============================
UDP_IP = "127.0.0.1"            # localhost
UDP_PORT_GUI = 5005             # Puerto destino GUI
UDP_PORT_LOGGER = 5006          # Puerto destino logger.py
UDP_PORT_FLASK = 6000           # Puerto destino app.py flask
TCP_PORT = 5010                 # Puerto para recepción de comandos (TCP)
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_socket.setblocking(False)   # No bloquear si no hay cliente
```

```
# --- Socket TCP para recepción de comandos desde el sitio web ---
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
tcp_socket.bind((UDP_IP, TCP_PORT))
tcp_socket.listen(1)
tcp_socket.setblocking(False)

# ==============================
# CALIBRACION SENSOR OXIGENO (dos puntos)
# ==============================
# --- Voltajes medidos durante calibracion (mV) y temperaturas (°C) ---
CAL1_V, CAL1_T = 1600, 25
CAL2_V, CAL2_T = 1300, 15

# --- Tabla de oxígeno saturado según temperatura (µg/L) ---
DO_Table = [
    14460, 14220, 13820, 13440, 13090, 12740, 12420, 12110, 11810, 11530,
    11260, 11010, 10770, 10530, 10300, 10080, 9860, 9660, 9460, 9270,
    9080, 8900, 8730, 8570, 8410, 8250, 8110, 7960, 7820, 7690,
    7560, 7430, 7300, 7180, 7070, 6950, 6840, 6730, 6630, 6530, 6410
]

# ==============================
# FUNCIONES SERIAL
# ==============================
def conectar_serial():
    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=0.1)
        time.sleep(3)
        print(f"[OK] Puerto serial abierto: {SERIAL_PORT} a {BAUD_RATE} bps")
        return ser
    except Exception as e:
        errores_activos.add(14)
        print(f"[ERROR] No se pudo abrir el puerto serial: {e}")
        return None

ser = None
while ser is None:
    ser = conectar_serial()
    if ser is None:
        print("[WARN] No hay dispositivo conectado. Reintentando en 5 segundos...")
        time.sleep(5)

def leer_linea_ultima():
    """
    Drena el buffer serial y devuelve SOLO la última línea válida disponible.
    Si no hay datos, retorna None sin bloquear el bucle.
    """
    global ser
    try:
        if ser.in_waiting == 0:
            return None  # nada que leer ahora

        last_line = None
        max_reads = 200
        reads = 0
```

```
        while ser.in_waiting > 0 and reads < max_reads:
            raw = ser.readline()
            if not raw:
                break
            linea = raw.decode('utf-8', errors='ignore').strip()
            if linea:
                last_line = linea
            reads += 1

        if not last_line:
            return None

        partes = last_line.split(',')
        if len(partes) != len(CANALES):
            return None

        valores = {canal: int(val) for canal, val in zip(CANALES, partes)}
        return valores

    except (serial.SerialException, OSError) as e:
        errores_activos.add(15)
        print(f"[ERROR] Puerto serial: {e}. Intentando reconectar...")
        try:
            ser.close()
        except:
            pass
        ser = None
        while ser is None:
            time.sleep(2)
            ser = conectar_serial()
        return None
    except Exception as e:
        errores_activos.add(16)
        print(f"[ERROR] leer_linea_ultima: {e}")
        return None

# ==============================
# FUNCIONES DE PROCESAMIENTO
# ==============================
def promedio_acumulador(acumulador, n_muestras):
    return {canal: suma / n_muestras for canal, suma in acumulador.items()}

def voltaje_acumulador(promedio):
    return {canal: val * CONVERSION_FACTOR[canal] for canal, val in promedio.items()}

def suavizado_exponencial(valor_actual, valor_anterior, alpha=ALPHA_SUAVIZADO):
    if valor_anterior is None:
        return valor_actual
    return alpha * valor_actual + (1 - alpha) * valor_anterior

# ==============================
# FUNCIONES DE TEMPERATURA
# ==============================
def leer_temperatura():
    """Lee la temperatura actual del DS18B20 en °C como float."""
    try:
```

```
        temperatura = sensor_ds18b20.get_temperature()
        return temperatura
    except Exception as e:
        errores_activos.add(17)
        print(f"[ERROR] No se pudo leer DS18B20: {e}")
        return 25.0

def temperatura_a_entero(temp):
    """Convierte la temperatura float a un entero entre 0 y 40 para indexar DO_Table."""
    temp_entero = int(round(temp))
    if temp_entero < 0:
        temp_entero = 0
    elif temp_entero > 40:
        temp_entero = 40
    return temp_entero

def leer_temperatura_cached():
    """Cache de temperatura para evitar bloqueos por lectura frecuente."""
    global _last_temp_read, _temp_cache
    ahora = time.time()
    if (ahora - _last_temp_read) >= TEMP_READ_INTERVAL:
        t = leer_temperatura()
        _temp_cache = t
        _last_temp_read = ahora
    return _temp_cache

# ==============================
# CONVERSION VOLTAJE → OXIGENO DISUELTO (mg/L)
# ==============================
def voltaje_a_DO(voltaje_mv, temperatura_c):
    # --- Limitar temperatura al rango de la tabla ---
    if temperatura_c < 0:
        temperatura_c = 0
    elif temperatura_c > 40:
        temperatura_c = 40

    # --- Voltaje de saturación interpolado según la temperatura ---
    V_saturacion = ((temperatura_c - CAL2_T) * (CAL1_V - CAL2_V) / (CAL1_T - CAL2_T)) + CAL2_V

    # --- Interpolación lineal de DO_Table usando temperatura decimal ---
    lower_index = int(temperatura_c)
    upper_index = min(lower_index + 1, 40)
    fraction = temperatura_c - lower_index

    DO_lower = DO_Table[lower_index]
    DO_upper = DO_Table[upper_index]
    DO_interp = DO_lower + (DO_upper - DO_lower) * fraction  # µg/L

    # --- Ajustar según voltaje ---
    DO_ug_L = voltaje_mv * DO_interp / V_saturacion
    DO_mg_L = DO_ug_L / 1000.0
    return DO_mg_L

# ==============================
# ACTUALIZAR PID (PID Y CONTROL)
# ==============================
def actualizar_pid(pH_compensado, OD_compensado):
```

```
    global pid_paused_ph, pid_paused_o2

    # --- pH subida ---
    error_up = setpoint_ph_up - pH_compensado
    if pid_paused_ph:
        pid_ph_up.auto_mode = False
        control_ph_up = 0
    else:
        if abs(error_up) <= DEADBAND_PH:
            pid_ph_up.auto_mode = False
            control_ph_up = 0
        elif error_up > 0:
            pid_ph_up.auto_mode = True
            pid_ph_down.auto_mode = False
            control_ph_up = pid_ph_up(pH_compensado)
        else:
            pid_ph_up.auto_mode = False
            control_ph_up = 0

    # --- pH bajada ---
    error_down = pH_compensado - setpoint_ph_down
    if pid_paused_ph:
        pid_ph_down.auto_mode = False
        control_ph_down = 0
    else:
        if abs(error_down) <= DEADBAND_PH:
            pid_ph_down.auto_mode = False
            control_ph_down = 0
        elif error_down > 0:
            pid_ph_down.auto_mode = True
            pid_ph_up.auto_mode = False
            control_ph_down = pid_ph_down(pH_compensado)
        else:
            pid_ph_down.auto_mode = False
            control_ph_down = 0

    # --- O2 ---
    if pid_paused_o2:
        pid_o2.auto_mode = False
        control_o2 = 0
    else:
        if abs(OD_compensado - setpoint_o2) <= DEADBAND_O2:
            pid_o2.auto_mode = False
            control_o2 = 0
        else:
            pid_o2.auto_mode = True
            control_o2 = pid_o2(OD_compensado)

    return control_ph_down, control_ph_up, control_o2

# ==============================
# CONFIGURACION PINS RELES
# ==============================
RELAY_PIN_PH_UP = 17    # GPIO para subir ph
RELAY_PIN_PH_DOWN = 27  # GPIO para bajar ph
RELAY_PIN_O2 = 25       # GPIO para controlar O2
# --- Pines para controlar el temporizador 555
```

```
GPIO_TRIGGER = 23 # Para disparar el temporizador 555
GPIO_RESET   = 24 # Para resetear el temporizador 555

GPIO.setmode(GPIO.BCM)
GPIO.setup(RELAY_PIN_PH_UP, GPIO.OUT)
GPIO.setup(RELAY_PIN_PH_DOWN, GPIO.OUT)
GPIO.setup(RELAY_PIN_O2, GPIO.OUT)

GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_RESET, GPIO.OUT)

# Estado inicial seguro (todo apagado / sin disparar)
GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
GPIO.output(RELAY_PIN_O2, GPIO.LOW)

# El 555 está en reposo con TRIGGER y RESET en HIGH
GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
GPIO.output(GPIO_RESET, GPIO.HIGH)


# ==============================
# LÓGICA 555 – FLANCOS PARA TRIGGER Y RESET
# ==============================

# Estados previos para detectar flancos (inicialmente LOW)
prev_ph_up  = False     # False = LOW, True = HIGH
prev_ph_down = False    # igual

# ==============================
# Funciones para enviar pulsos a TRIGGER y RESET
# ==============================
def pulso_trigger():
    """Envía un pulso corto al TRIGGER (flanco descendente)."""
    GPIO.output(GPIO_TRIGGER, GPIO.LOW)
    time.sleep(0.03)
    GPIO.output(GPIO_TRIGGER, GPIO.HIGH)


def pulso_reset():
    """Envía un pulso corto al RESET (flanco descendente)."""
    GPIO.output(GPIO_RESET, GPIO.LOW)
    time.sleep(0.03)
    GPIO.output(GPIO_RESET, GPIO.HIGH)
pulso_reset() # Resetea al iniciar

def logica_temporizador_555(ph_up_on, ph_down_on):
    """
    Detecta flancos ascendentes y descendentes en pH Up y pH Down.
    Maneja TRIGGER y RESET del 555 según la lógica real del circuito.
    """
    global prev_ph_up, prev_ph_down

    # --- Señales actuales ---
    current_up = ph_up_on     # True o False
    current_down = ph_down_on
```

```
    # ======================================
    #   1) FLANCO ASCENDENTE → RESET
    # ======================================
    if current_up and not prev_ph_up:
        pulso_reset()

    if current_down and not prev_ph_down:
        pulso_reset()

    # ======================================
    #   2) FLANCO DESCENDENTE → TRIGGER
    # ======================================
    if prev_ph_up and not current_up:
        pulso_trigger()

    if prev_ph_down and not current_down:
        pulso_trigger()

    # Actualizar estados previos
    prev_ph_up = current_up
    prev_ph_down = current_down


# ==============================
# CONFIGURACIÓN DEL CONTROL MANUAL Y SEGURIDAD DE pH
# ==============================
# --- Parámetros ajustables ---
CAUDAL_BOMBA_ML_MIN = 70.0                  # Caudal nominal de las bombas (ml/min)
TIEMPO_MAX_O2 = 18000.0                     # Tiempo máximo de activación manual del O2 (segundos)
DOSIFICACIONES_PH_UP = [10.0, 20.0, 40.0]   # Volúmenes predefinidos para subir el pH (ml)
DOSIFICACIONES_PH_DOWN = [10.0, 20.0, 40.0] # Volúmenes predefinidos para bajar el pH (ml)
COOLDOWN_PH_SEG = 3600                      # Tiempo mínimo entre dosificaciones (1 hora)

# --- Límites de variación permitidos ---
DELTA_PH_MAX_HORA = 0.10                    # Máximo cambio permitido por hora
DELTA_PH_MAX_DIA = 0.50                     # Máximo cambio permitido por día

# --- Variables de estado ---
tareas_manual = []                          # Lista de acciones manuales pendientes o en curso
ultimo_ph_up = 0.0                          # Hora (timestamp) de la última dosificación pH↑
ultimo_ph_down = 0.0                        # Hora (timestamp) de la última dosificación pH↓
bloqueo_seguridad_ph = False                # True = Bloqueo por exceso de variación de pH
bloqueo_ph_hasta = 0                        # timestamp futuro para desbloquear
bloqueo_ph_tipo = ""                        # puede tomar valores como hora o dia

# ==============================
# CICLOS Y UMBRALES DE ACTUACION (en segundos)
# ==============================
CICLO_PH_UP = 2.0
CICLO_PH_DOWN = 2.0
CICLO_O2 = 2.0

MIN_ACTUACION_PH_UP = 0.2
MIN_ACTUACION_PH_DOWN = 0.2
MIN_ACTUACION_O2 = 0.2

inicio_ciclo_ph_up = time.time()
```

```
inicio_ciclo_ph_down = time.time()
inicio_ciclo_o2 = time.time()

def pin_en_tarea_manual(pin):
    """
    Devuelve True si el pin está siendo usado por alguna tarea manual
    (pH↑, pH↓ u O2).
    """
    return any(t["pin"] == pin for t in tareas_manual)


def control_por_tiempo(salida_pid, inicio_ciclo, ciclo, pin, min_actuacion=0.0):
    """
    Controla el relé por tiempo proporcional a la salida PID.
    Devuelve el nuevo inicio de ciclo y el tiempo de activación (s).
    """
    try:
        tiempo_on = (salida_pid / 100.0) * ciclo
        if tiempo_on < min_actuacion:
            tiempo_on = 0
        tiempo_transcurrido = time.time() - inicio_ciclo
        if tiempo_transcurrido >= ciclo:
            inicio_ciclo += ciclo
            tiempo_transcurrido -= ciclo
        if tiempo_transcurrido < tiempo_on:
            GPIO.output(pin, GPIO.HIGH)
        else:
            GPIO.output(pin, GPIO.LOW)
        return inicio_ciclo, tiempo_on
    except Exception as e:
        errores_activos.add(19)
        print(f"[ERROR GPIO] {e}")
        return inicio_ciclo, 0

# ============================
# PROCESAR LECTURAS
# ============================
def procesar_lectura(lectura, acumulador, contador):
    for canal in CANALES:
        acumulador[canal] += lectura[canal]
    contador += 1
    if contador >= N_MUESTRAS:
        promedio = promedio_acumulador(acumulador, N_MUESTRAS)
        acumulador = {canal: 0 for canal in CANALES}
        contador = 0
        return promedio, True, acumulador, contador
    return None, False, acumulador, contador

# ============================
# CONVERTIR Y COMPENSAR LECTURAS
# ============================
def convertir_y_compensar(promedio, voltaje_suavizado):
    voltaje = voltaje_acumulador(promedio)
    for canal in CANALES:
        voltaje_suavizado[canal] = suavizado_exponencial(
            voltaje[canal],
            voltaje_suavizado[canal]
```

```
        )
    voltaje_a0 = voltaje_suavizado['A0']
    voltaje_a1 = voltaje_suavizado['A1']
    temperatura_float = leer_temperatura_cached()
    temperatura_actual = temperatura_a_entero(temperatura_float)
    OD_compensado = voltaje_a_DO(voltaje_a1 * 1000, temperatura_float)
    T_kelvin = temperatura_actual + 273.15
    a25 = -5.7
    b = 21.34
    aT = a25 * (T_kelvin / 298.15)
    pH_compensado = aT * voltaje_a0 + b
    return voltaje_a0, voltaje_a1, pH_compensado, OD_compensado, temperatura_float, voltaje_suavizado

# ==============================
# MOSTRAR DATOS EN CONSOLA
# ==============================
def mostrar_datos_consola(voltaje_a0, voltaje_a1, pH_compensado, OD_compensado,
                          control_ph_down, control_ph_up, control_o2,
                          pid_ph_up, pid_ph_down, pid_o2, temperatura_float, ser, errores_udp_str):
    p_up, i_up, d_up = pid_ph_up.components
    p_down, i_down, d_down = pid_ph_down.components
    p_o2, i_o2, d_o2 = pid_o2.components
    '''print(
        f"a0:{voltaje_a0:.2f},pH:{pH_compensado:.2f},a1:{voltaje_a1:.2f},O2:{OD_compensado:.2f},"
        f"o_pH_down:{control_ph_down:.2f},o_pH_up:{control_ph_up:.2f},o_O2:{control_o2:.2f}"
        f"[pH UP] P:{p_up:.2f},I:{i_up:.2f},D:{d_up:.2f}"
        f"[pH DOWN] P:{p_down:.2f},I:{i_down:.2f},D:{d_down:.2f}"
        f"[O2] P:{p_o2:.2f},I:{i_o2:.2f},D:{d_o2:.2f},"
        f"T:{temperatura_float:.1f}°C,Error:{errores_udp_str}"
    )'''
    #print(f"[DEBUG] in_waiting={ser.in_waiting}")

    print(
            f"a0:{voltaje_a0:.2f},pH:{pH_compensado:.2f},a1:{voltaje_a1:.2f},O2:{OD_compensado:.2f},"
            f"o_pH_down:{control_ph_down:.2f},o_pH_up:{control_ph_up:.2f},o_O2:{control_o2:.2f},"
            f"T:{temperatura_float:.1f}°C,Error:{errores_udp_str},"
        )

# ==============================
# ENVIAR DATOS VIA UDP
# ==============================
def enviar_datos_udp(voltaje_a0, voltaje_a1, pH_compensado, OD_compensado,
                     control_ph_down, control_ph_up, control_o2,
                     pid_ph_up, pid_ph_down, pid_o2, temperatura_float,
                     tiempo_on_up, tiempo_on_down, tiempo_on_o2, errores_udp_str):
    """Envía los datos en formato CSV mediante UDP, incluyendo tiempos de activación."""
    p_up, i_up, d_up = pid_ph_up.components
    p_down, i_down, d_down = pid_ph_down.components
    p_o2, i_o2, d_o2 = pid_o2.components

    mensaje = (
        f"{voltaje_a0:.4f},{pH_compensado:.4f},{voltaje_a1:.4f},{OD_compensado:.4f},"
        f"{control_ph_down:.2f},{control_ph_up:.2f},{control_o2:.2f},"
        f"{p_down:.4f},{i_down:.4f},{d_down:.4f},"
        f"{p_up:.4f},{i_up:.4f},{d_up:.4f},"
        f"{p_o2:.4f},{i_o2:.4f},{d_o2:.4f},"
        f"{temperatura_float:.2f},"
```

```
        f"{tiempo_on_down:.3f},{tiempo_on_up:.3f},{tiempo_on_o2:.3f},"
        f"{errores_udp_str}"
    ).encode("utf-8")

    try:
        udp_socket.sendto(mensaje, (UDP_IP, UDP_PORT_GUI))      # Enviar a GUI
        udp_socket.sendto(mensaje, (UDP_IP, UDP_PORT_LOGGER))   # Enviar al registrador
    except Exception as e:
        errores_activos.add(18)
        print(f"[WARN UDP] No se pudo enviar el paquete: {e}")
# ==============================
# ENVIAR DATOS VIA UDP A FLASK
# ==============================
def enviar_estado_udp_flask(estado_json):
    try:
        udp_socket.sendto(estado_json.encode(), (UDP_IP, UDP_PORT_FLASK))
    except Exception as e:
        print("[WARN UDP → Flask]", e)

def notificar_flask(tipo, mensaje):
    """
    Envía una notificación al servidor Flask de manera segura.
    - tipo: str   → 'seguridad', 'reinicio', 'info', etc.
    - mensaje: str → texto que aparecerá en notificaciones
    """
    try:
        requests.post(
            "http://127.0.0.1:5000/api/notificacion_sistema",
            json={"tipo": tipo, "mensaje": mensaje},
            timeout=0.5
        )
    except Exception as e:
        print("[WARN] No se pudo notificar a Flask:", e)


# ==============================
# RECEPCIÓN DE COMANDOS TCP [EXTENDIDA]
# ==============================
def escuchar_confirmacion_tcp():
    """
    Escucha comandos externos por TCP para reanudar o pausar PIDs,
    y ejecutar acciones manuales seguras mediante códigos numéricos.
      - 1 = reanudar PID pH
      - 2 = reanudar PID O2
      - 3 = reanudar ambos
      - 4 = desactivar PID pH
      - 5 = desactivar PID O2
      - 6 = detener todos los actuadores manuales
      - 7 <tiempo> = activar aireadores (O2) por <tiempo> segundos
      - 8 <preset> = dosificar pH↑ (según volumen predefinido)
      - 9 <preset> = dosificar pH↓ (según volumen predefinido)
    """
    global pid_paused_ph, pid_paused_o2                      # estados de pausa de PIDs
    global ultimo_ph_up, ultimo_ph_down, bloqueo_seguridad_ph # cooldown y bloqueo seguridad
    global prev_ph_down, prev_ph_up                     # estados previos para 555

    try:
```

```
conn = None
try:
    conn, addr = tcp_socket.accept()   # Acepta conexión entrante (no bloqueante)
    conn.settimeout(1.0)               # Recv con timeout para evitar bloqueos
    data = conn.recv(1024)             # Recibe el comando enviado por el cliente
    if not data:                       # Si no hay datos, cerrar conexión
        conn.close()
        return
    comando = data.decode().strip()    # Convierte los bytes a texto
    conn.close()                       # Cierra la conexión TCP luego de recibir el comando
except BlockingIOError:
    return

partes = comando.split()               # Divide el comando en partes (separado por espacios)
codigo = partes[0]                     # Toma el codigo de comando
valor = partes[1] if len(partes) > 1 else None # Toma el valor adicional si existe
ahora = time.time()                    # tiempo actual

# --- Reanudar PIDs ---
if codigo == "1":
      if bloqueo_seguridad_ph:
          print("[TCP] Ignorado '1' → bloqueo de seguridad pH activo")
          return
      pid_paused_ph = False
      print("[TCP] Reanudado PID pH (sin bloqueo)")


elif codigo == "2":
    pid_paused_o2 = False
    print("[TCP] Reanudado PID O2")

elif codigo == "3":
    if bloqueo_seguridad_ph:
        print("[TCP] Ignorado '3' → bloqueo seguridad pH activo")
        pid_paused_o2 = False  # pero sí permitimos reactivar O2
        return
    pid_paused_ph = False
    pid_paused_o2 = False
    print("[TCP] Reanudados ambos PID")


# 4 = Desactivar PID de pH
elif codigo == "4":
    pid_ph_up.auto_mode = False
    pid_ph_down.auto_mode = False
    pid_paused_ph = True
    print("[TCP] PID de pH desactivado manualmente.")

# 5 = Desactivar PID de O2
elif codigo == "5":
    pid_o2.auto_mode = False
    pid_paused_o2 = True
    print("[TCP] PID de O2 desactivado manualmente.")

# 6 = STOP (parcial o total)
elif codigo == "6":
```

```
# Si viene con un parámetro, decidir acción
modo = str(valor) if valor is not None else "0"

# ----------------------------
# 6 1 → Detener SOLO aireador
# ----------------------------
if modo == "1":
    GPIO.output(RELAY_PIN_O2, GPIO.LOW)
# eliminar tareas O2 activas
    tareas_manual[:] = [t for t in tareas_manual if t["tipo"] != "O2"]
    print("[MANUAL] Aireadores detenidos (6 1).")
    return

# ----------------------------
# 6 2 → PARADA DE EMERGENCIA
# ----------------------------
if modo == "2":
    GPIO.output(RELAY_PIN_O2, GPIO.LOW)
    GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
    GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
    GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
    pulso_reset()
    prev_ph_up = False
    prev_ph_down = False

    tareas_manual.clear()

    # Mantener PIDs en manual pero quietos
    pid_o2.auto_mode = False
    pid_ph_up.auto_mode = False
    pid_ph_down.auto_mode = False

    pid_paused_o2 = True
    pid_paused_ph = True

    notificar_flask("seguridad", "🛑 PARADA DE EMERGENCIA ACTIVADA")

    print("[EMERGENCIA] Parada TOTAL — todos los actuadores apagados (6 2).")
    return

# Si el usuario manda solo "6" → tratar como parada total
GPIO.output(RELAY_PIN_O2, GPIO.LOW)
GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
pulso_reset()

prev_ph_up = False
prev_ph_down = False

tareas_manual.clear()
pid_o2.auto_mode = False
pid_ph_up.auto_mode = False
pid_ph_down.auto_mode = False

pid_paused_o2 = True
pid_paused_ph = True
```

```
            print("[MANUAL] '6' recibido sin parámetro → parada total por seguridad.")


        # 7 = Activar O2 manualmente por <tiempo> segundos
        elif codigo == "7" and valor:               # Formato de recepción "7 30" para 30 segundos
            try:
                tiempo = float(valor)               # Convierte valor a float
            except:
                print("[MANUAL] Valor de tiempo inválido para O2.")
                return
            tiempo = min(max(tiempo, 0.0), TIEMPO_MAX_O2) # limitar al máximo
            pid_o2.auto_mode = False                    # desactivar PID O2
            pid_paused_o2 = True                        # pausar PID O2
            if any(t["pin"] == RELAY_PIN_O2 for t in tareas_manual): # Verificar si ya hay una tarea
activa
                print("[MANUAL] O2 ya tiene una tarea activa. Comando ignorado.")
                return
            tareas_manual.append({
                "tipo": "O2",                           # tipo de tarea
                "pin": RELAY_PIN_O2,                    # pin a activar
                "t_fin": ahora + tiempo                 # tiempo de finalización (tiempo actual +
duración)
            })
            print(f"[MANUAL] Aireador activado durante {tiempo:.1f} segundos.")

        # 8 = Dosificación pH↑ (preset)
        elif codigo == "8" and valor: # formato de recepción "8 1" para preset 1
            if bloqueo_seguridad_ph:  # Verificar si el sistema está bloqueado por seguridad
                print("[MANUAL] Bloqueo de seguridad pH activo. Comando rechazado.")
                return
            try:
                preset = int(valor)   # convertir a entero
            except:
                print("[MANUAL] Preset no válido para pH↑.")
                return
            if not (0 <= preset < len(DOSIFICACIONES_PH_UP)):   # Verificar que el numero de preset esté
en rango
                print("[MANUAL] Preset fuera de rango para pH↑.")
                return
            if ahora - ultimo_ph_up < COOLDOWN_PH_SEG:          # Verificar cooldown
                espera = COOLDOWN_PH_SEG - (ahora - ultimo_ph_up)
                print(f"[MANUAL] pH↑ en enfriamiento ({espera/60:.1f} min restantes).")
                return

            # 🔒 Cooldown cruzado – si pH↓ está en cooldown también bloquear
            if ahora - ultimo_ph_down < COOLDOWN_PH_SEG:
                espera = COOLDOWN_PH_SEG - (ahora - ultimo_ph_down)
                print(f"[MANUAL] pH↑ bloqueado por cooldown de pH↓ ({espera/60:.1f} min restantes).")
                return

            ml = DOSIFICACIONES_PH_UP[preset]          # volumen a dosificar (ml)
            duracion = ml_a_segundos(ml)               # Convertir ml a segundos
            pid_ph_up.auto_mode = False                # Desactivar PID de pH↑
            pid_ph_down.auto_mode = False              # Desactivar PID de pH↓
            pid_paused_ph = True                       # Pausar PID de pH
```

```
            if any(t["pin"] == RELAY_PIN_PH_UP for t in tareas_manual): # Verificar si ya hay una tarea activa
                print("[MANUAL] pH↑ ya tiene una tarea activa. Comando ignorado.")
                return
            tareas_manual.append({
                "tipo": "pH↑",                          # Tipo de tarea
                "pin": RELAY_PIN_PH_UP,                 # Pin a activar
                "t_fin": ahora + duracion               # Tiempo de finalización
            })
            ultimo_ph_up = ahora                        # Actualizar tiempo de última dosificación
            print(f"[MANUAL] pH↑ preset {preset} → {ml:.1f} ml ({duracion:.1f}s).")

        # 9 = Dosificación pH↓ (analogo al anterior)
        elif codigo == "9" and valor:
            if bloqueo_seguridad_ph:
                print("[MANUAL] Bloqueo de seguridad pH activo. Comando rechazado.")
                return
            try:
                preset = int(valor)
            except:
                print("[MANUAL] Preset no válido para pH↓.")
                return
            if not (0 <= preset < len(DOSIFICACIONES_PH_DOWN)):
                print("[MANUAL] Preset fuera de rango para pH↓.")
                return
            if ahora - ultimo_ph_down < COOLDOWN_PH_SEG:
                espera = COOLDOWN_PH_SEG - (ahora - ultimo_ph_down)
                print(f"[MANUAL] pH↓ en enfriamiento ({espera/60:.1f} min restantes).")
                return

            # Bloqueo cruzado: si pH↑ está en cooldown, bloquear pH↓ también
            if ahora - ultimo_ph_up < COOLDOWN_PH_SEG:
                espera = COOLDOWN_PH_SEG - (ahora - ultimo_ph_up)
                print(f"[MANUAL] pH↓ bloqueado por cooldown de pH↑ ({espera/60:.1f} min restantes).")
                return


            ml = DOSIFICACIONES_PH_DOWN[preset]
            duracion = ml_a_segundos(ml)
            pid_ph_up.auto_mode = False
            pid_ph_down.auto_mode = False
            pid_paused_ph = True
            if any(t["pin"] == RELAY_PIN_PH_DOWN for t in tareas_manual):
                print("[MANUAL] pH↓ ya tiene una tarea activa. Comando ignorado.")
                return
            tareas_manual.append({
                "tipo": "pH↓",
                "pin": RELAY_PIN_PH_DOWN,
                "t_fin": ahora + duracion
            })
            ultimo_ph_down = ahora
            print(f"[MANUAL] pH↓ preset {preset} → {ml:.1f} ml ({duracion:.1f}s).")

        # 10 = Reiniciar sistema de seguridad pH
        elif codigo == "10":
            print("[TCP] Reinicio manual del bloqueo de seguridad pH")
```

```
            bloqueo_seguridad_ph = False
            pid_paused_ph = False
            bloqueo_ph_hasta = 0
            bloqueo_ph_tipo = ""

            # Reset diario y base
            monitorear_seguridad_ph.variacion_acumulada = 0.0
            monitorear_seguridad_ph.ph_base = None  # se recalculará en la próxima lectura
            monitorear_seguridad_ph.fecha_actual = datetime.now().strftime("%Y-%m-%d")

            ultimo_ph_down = 0
            ultimo_ph_up = 0

            # También resetear tareas o reset
            pulso_reset()  # importante para sincronizar relés/555

            # Notificación a Flask
            notificar_flask("seguridad", "🟢 PID pH reiniciado manualmente por el usuario")

    except Exception as e:
        print(f"[WARN TCP] {e}")

# ==============================
# ACTUALIZAR ACTUADORES
# ==============================
def actualizar_actuadores(control_ph_up, control_ph_down, control_o2,
                          inicio_ciclo_ph_up, inicio_ciclo_ph_down, inicio_ciclo_o2):
    """
    Actualiza los relés de pH↑, pH↓ y O2 según:
      - Salidas PID (modo automático)
      - Tareas manuales (si las hay, tienen prioridad y el PID NO toca ese pin)
    Además, actualiza la lógica del temporizador 555 en base al estado real
    de los relés de pH.
    """

    # ---- pH UP ----
    if not pin_en_tarea_manual(RELAY_PIN_PH_UP):
        # Solo si NO hay tarea manual para este pin, aplica el PWM del PID
        inicio_ciclo_ph_up, tiempo_on_up = control_por_tiempo(
            control_ph_up,
            inicio_ciclo_ph_up,
            CICLO_PH_UP,
            RELAY_PIN_PH_UP,
            MIN_ACTUACION_PH_UP
        )
    else:
        # Hay una tarea manual activa → NO tocar el pin desde el PID
        tiempo_on_up = 0.0

    # ---- pH DOWN ----
    if not pin_en_tarea_manual(RELAY_PIN_PH_DOWN):
        inicio_ciclo_ph_down, tiempo_on_down = control_por_tiempo(
            control_ph_down,
            inicio_ciclo_ph_down,
            CICLO_PH_DOWN,
            RELAY_PIN_PH_DOWN,
            MIN_ACTUACION_PH_DOWN
```

```
        )
    else:
        tiempo_on_down = 0.0

    # ---- O2 ----
    if not pin_en_tarea_manual(RELAY_PIN_O2):
        inicio_ciclo_o2, tiempo_on_o2 = control_por_tiempo(
            control_o2,
            inicio_ciclo_o2,
            CICLO_O2,
            RELAY_PIN_O2,
            MIN_ACTUACION_O2
        )
    else:
        tiempo_on_o2 = 0.0

    # ------------------------------------------
    #  LÓGICA DEL 555 (se ejecuta SIEMPRE)
    #  - Lee el estado real actual de los relés, ya
    #    sea por manual o por automático.
    # ------------------------------------------
    ph_up_on = GPIO.input(RELAY_PIN_PH_UP) == GPIO.HIGH
    ph_down_on = GPIO.input(RELAY_PIN_PH_DOWN) == GPIO.HIGH

    logica_temporizador_555(ph_up_on, ph_down_on)

    # Retornar tiempos (como antes)
    return (
        inicio_ciclo_ph_up,
        inicio_ciclo_ph_down,
        inicio_ciclo_o2,
        tiempo_on_up,
        tiempo_on_down,
        tiempo_on_o2,
    )

# ==============================
# FUNCIONES AUXILIARES DEL CONTROL MANUAL Y SEGURIDAD DE pH [NUEVO]
# ==============================
def ml_a_segundos(ml):
    """
    Convierte una cantidad de mililitros en el tiempo (segundos)
    que debe permanecer activa la bomba, en base al caudal nominal.
    """
    return (ml / CAUDAL_BOMBA_ML_MIN) * 60.0

def monitorear_seguridad_ph(ph_actual):
    """
    Sistema de seguridad continuo:
      - Detecta variaciones bruscas respecto a un pH_base.
      - Cada variación >= DELTA_PH_MAX_HORA → bloqueo 1h.
      - Suma variaciones al acumulado diario.
      - Si acumulado diario >= DELTA_PH_MAX_DIA → bloqueo 24h.
      - Reset diario EXACTAMENTE a medianoche.
    """

    global bloqueo_seguridad_ph, bloqueo_ph_hasta, bloqueo_ph_tipo
```

```
    global pid_paused_ph

    ahora = time.time()
    fecha_hoy = datetime.now().strftime("%Y-%m-%d")

    # ------------------------------------------------------------
    # Inicialización persistente (solo primera llamada)
    # ------------------------------------------------------------
    if not hasattr(monitorear_seguridad_ph, "ph_base"):
        monitorear_seguridad_ph.ph_base = ph_actual
        monitorear_seguridad_ph.fecha_actual = fecha_hoy
        monitorear_seguridad_ph.variacion_acumulada = 0.0
        return

    # Si ph_base fue reiniciado manualmente → recalcular base
    if monitorear_seguridad_ph.ph_base is None:
        monitorear_seguridad_ph.ph_base = ph_actual
        return

    # ------------------------------------------------------------
    # Calcular variación respecto a pH_base
    # ------------------------------------------------------------
    delta = abs(ph_actual - monitorear_seguridad_ph.ph_base)
    print("delta:", delta, "ph_base:", monitorear_seguridad_ph.ph_base)

    # ------------------------------------------------------------
    # RESET DIARIO A MEDIANOCHE
    # ------------------------------------------------------------
    if monitorear_seguridad_ph.fecha_actual != fecha_hoy:
        print("Reset diario automático del acumulado pH")
        monitorear_seguridad_ph.fecha_actual = fecha_hoy
        monitorear_seguridad_ph.variacion_acumulada = 0.0
        notificar_flask("seguridad", "🔄 PID pH reanudado automáticamente por reinicio diario")

    # ------------------------------------------------------------
    # Si está bloqueado → verificar si ya terminó el bloqueo
    # ------------------------------------------------------------
    if bloqueo_seguridad_ph:
        if ahora >= bloqueo_ph_hasta:
            print("Bloqueo pH finalizado → reactivando PID")
            bloqueo_seguridad_ph = False
            pid_paused_ph = False

            notificar_flask("seguridad", "🟢 PID pH reanudado automáticamente al finalizar el bloqueo")

            # El pH actual pasa a ser el nuevo pH base
            monitorear_seguridad_ph.ph_base = ph_actual

        return

    # ------------------------------------------------------------
    # 1) Variación brusca → bloqueo 1 hora
    # ------------------------------------------------------------
    if delta >= DELTA_PH_MAX_HORA:
        print(f"Variación brusca: ΔpH={delta:.2f} → bloqueo 1h")

        bloqueo_seguridad_ph = True
```

```
        bloqueo_ph_hasta = ahora + 3600  # 1 hora
        bloqueo_ph_tipo = "hora"
        pid_paused_ph = True

        notificar_flask("seguridad", "⚠ PID pH bloqueado 1 hora por variación brusca del pH")

        # Sumar variación al acumulado diario
        monitorear_seguridad_ph.variacion_acumulada += delta

        return

    # ----------------------------------------------------------
    # 2) Variación acumulada diaria → bloqueo 24h
    # ----------------------------------------------------------
    if monitorear_seguridad_ph.variacion_acumulada >= DELTA_PH_MAX_DIA:
        print(f"Límite diario superado: Δacum={monitorear_seguridad_ph.variacion_acumulada:.2f} → bloqueo
24h")

        bloqueo_seguridad_ph = True
        bloqueo_ph_hasta = ahora + 86400  # 24 horas
        bloqueo_ph_tipo = "dia"
        pid_paused_ph = True

        notificar_flask("seguridad", "⛔ PID pH bloqueado 24 horas por exceso de variación diaria")


        return


# ==============================
# PROCESADOR DE TAREAS MANUALES (NO BLOQUEANTE) [NUEVO]
# ==============================
def procesar_tareas_manuales(tiempo_actual):
    """
    Supervisa las tareas manuales activas.
    Si el tiempo de activación ha finalizado, apaga el relé correspondiente.
    Mientras una tarea esté activa, el PID asociado permanece desactivado.
    """
    global tareas_manual

    tareas_activas = []          # Lista temporal para mantener solo las tareas en curso

    for tarea in tareas_manual:  # Recorre las tareas activas
        tipo = tarea["tipo"]     # 'pH_UP', 'pH_DOWN' o 'O2'
        pin = tarea["pin"]       # Pin GPIO asociado
        fin = tarea["t_fin"]     # Tiempo de finalización

        # Si el relé aún no está encendido
        '''
        Solo se ejecuta una vez por tarea en el primer ciclo despues de
        crearla, en los siguientes ciclos el rele ya esta activado, asi
        que esta linea se salta automaticamente
        '''
        if GPIO.input(pin) == GPIO.LOW:
            GPIO.output(pin, GPIO.HIGH)
            print(f"[MANUAL] {tipo} activado.")
```

```
        # Verifica si ya transcurrió el tiempo asignado
        '''
        Si ya paso el tiempo, se apaga el rele y se imprime finalizado
        '''
        if tiempo_actual >= fin:
            GPIO.output(pin, GPIO.LOW)
            print(f"[MANUAL] {tipo} finalizado.")
        else:
            tareas_activas.append(tarea) # si todavia no terminó se vuelve a guardar en tareas_activas

    tareas_manual = tareas_activas  # Actualiza la lista con las tareas que siguen en ejecución

# ==============================
# GESTIONAR PIDS SEGUN ERRORES
# ==============================
def gestionar_pids_por_error(errores_activos):
    global pid_paused_ph, pid_paused_o2
    global prev_ph_up, prev_ph_down
    errores_ph = {1, 4, 7, 11}
    errores_o2 = {2, 5, 8, 12}
    errores_temp = {3, 6, 9, 10}

    # --- pH ---
    if errores_activos & errores_ph:
        pid_ph_up.auto_mode = False
        pid_ph_down.auto_mode = False
        GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
        GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
        GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
        pulso_reset()
        prev_ph_up = False
        prev_ph_down = False
        pid_paused_ph = True
    elif not pid_paused_ph:
        pid_ph_up.auto_mode = True
        pid_ph_down.auto_mode = True

    # --- O2 ---
    if errores_activos & errores_o2:
        pid_o2.auto_mode = False
        GPIO.output(RELAY_PIN_O2, GPIO.LOW)
        GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
        pulso_reset()
        prev_ph_down = False
        prev_ph_up = False
        pid_paused_o2 = True
    elif not pid_paused_o2:
        pid_o2.auto_mode = True

    # --- Temperatura ---
    if errores_activos & errores_temp:
        pid_ph_up.auto_mode = False
        pid_ph_down.auto_mode = False
        pid_o2.auto_mode = False
        GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
        GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
        GPIO.output(RELAY_PIN_O2, GPIO.LOW)
```

```
        GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
        pulso_reset()
        prev_ph_up = False
        prev_ph_down = False
        pid_paused_ph = True
        pid_paused_o2 = True

# ==============================
# DETECCIÓN DE FALLOS EN SENSORES
# (inválidos → rango → congelado → fluctuación → coherencia)
# ==============================
codigo_error_actual = 0                                  # 0 = sin error

def _es_num(x):                                          # verifica si x es un número válido (no None, no
NaN)
    return isinstance(x, (int, float)) and not (x != x)  # descarta NaN

def verificar_sensores(pH, OD, T):
    """
    Retorna un conjunto de códigos de error activos (pudiendo coexistir varios).
    Ejemplo: {4, 5} si fallan simultáneamente los sensores de pH y O2.
    Mecanismos:
      1) Rango físico válido
      2) Lecturas inválidas consecutivas (None/NaN)
      3) Valor "congelado" (solo pH y O2; T excluida para evitar falsos positivos)
      4) Coherencia física básica: T↑ y O2↑ (>10%) simultáneo
      5) None inmediato cuenta como inválido
      6) Fluctuación anómala (sensor desconectado)
    """
    errores = set()                    # conjunto de errores activos
    st = verificar_sensores.__dict__   # almacena estado entre llamadas

    # --- Límites físicos ---
    PH_MIN, PH_MAX = 0.0, 14.0  # rango pH válido
    O2_MIN, O2_MAX = 0.0, 20.0  # rango O2 en mg/L
    T_MIN,  T_MAX  = 0.0, 40.0  # rango T en °C

    # --- Parámetros de estabilidad / consecutivos ---
    N_MAX_FALLAS = 4            # Inválidos consecutivos
    N_HIST, N_HIST_T = 50, 200  # Ventana histórica para detectar "congelado" ( pH/O2 , T )
    VAR_MIN_PH = 0.001          # Variación mínima esperada en pH
    VAR_MIN_O2 = 0.001          # Variación mínima esperada en O2
    VAR_MIN_T = 0.001           # Variación mínima esperada en T
    FLUC_UMBRAL_PH = 1.2        # Umbral de desviación estándar para detectar fluctuación anómala en pH
    FLUC_UMBRAL_O2 = 1.2        # Umbral de desviación estándar para detectar fluctuación anómala en O2

    # --- Estado persistente (historial y contadores) ---
    if "hist" not in st:                            # inicialización única
        st["hist"] = {"pH": [], "O2": [], "T": []}  # historial de lecturas
        st["bad"]  = {"pH": 0,  "O2": 0,  "T": 0}   # contadores de inválidos consecutivos

    # --- Mecanismo 5 + 2: None/NaN e inválidos consecutivos ---
    for name, val in (("pH", pH), ("O2", OD), ("T", T)):
        if not _es_num(val):             # None o NaN -> suma contador
            st["bad"][name] += 1         # contador de inválidos consecutivos
        else:
            st["bad"][name] = 0          # resetea contador si es válido
```

```
    if st["bad"]["pH"] >= N_MAX_FALLAS:
        errores.add(4)                  # pH inválidos consecutivos
    if st["bad"]["O2"] >= N_MAX_FALLAS:
        errores.add(5)                  # O2 inválidos consecutivos
    if st["bad"]["T"] >= N_MAX_FALLAS:
        errores.add(6)                  # T inválidos consecutivos

    # Si alguno es None/NaN pero aún no llegó al umbral, no seguimos evaluando ese canal
    if not (_es_num(pH) and _es_num(OD) and _es_num(T)):
        if not _es_num(pH): errores.add(4)
        if not _es_num(OD): errores.add(5)
        if not _es_num(T):  errores.add(6)
        return errores

    # --- Mecanismo 1: Rango físico ---
    if not (PH_MIN <= pH <= PH_MAX):
        errores.add(1)  # pH fuera de rango
    if not (O2_MIN <= OD <= O2_MAX):
        errores.add(2)  # O2 fuera de rango
    if not (T_MIN <= T <= T_MAX):
        errores.add(3)  # T fuera de rango

    # --- Mecanismo 3: Valor "congelado" (solo pH y O2; T excluida) ---
    st["hist"]["pH"].append(pH)
    st["hist"]["O2"].append(OD)
    st["hist"]["T"].append(T)
    if len(st["hist"]["pH"]) > N_HIST: st["hist"]["pH"].pop(0)
    if len(st["hist"]["O2"]) > N_HIST: st["hist"]["O2"].pop(0)
    if len(st["hist"]["T"])  > N_HIST_T: st["hist"]["T"].pop(0)

    if len(st["hist"]["pH"]) == N_HIST:
        if (max(st["hist"]["pH"]) - min(st["hist"]["pH"])) < VAR_MIN_PH: # se calcula la diferencia max-
min
            errores.add(7)  # pH "congelado"

    if len(st["hist"]["O2"]) == N_HIST:
        if (max(st["hist"]["O2"]) - min(st["hist"]["O2"])) < VAR_MIN_O2: # se calcula la diferencia max-
min
            errores.add(8)  # O2 "congelado"

    if len(st["hist"]["T"]) == N_HIST_T:
        if (max(st["hist"]["T"]) - min(st["hist"]["T"])) < VAR_MIN_T:    # se calcula la diferencia max-
min
            errores.add(9)  # T "congelada"

    # --- Mecanismo 6: Fluctuación anómala (sensor desconectado) ---
    # Si la desviación estándar en la ventana supera un umbral, se asume entrada flotante
    try:
        if len(st["hist"]["pH"]) >= 10:
            std_ph = statistics.pstdev(st["hist"]["pH"])
            if std_ph > FLUC_UMBRAL_PH:     # Umbral de fluctuación anómala (ajustable)
                errores.add(11)             # Fluctuación anómala pH

        if len(st["hist"]["O2"]) >= 10:
            std_o2 = statistics.pstdev(st["hist"]["O2"])
            if std_o2 > FLUC_UMBRAL_O2:     # Umbral de fluctuación anómala (ajustable)
```

```
                errores.add(12)             # Fluctuación anómala O2
        except statistics.StatisticsError:
            pass

        # --- Mecanismo 4: Coherencia física básica ---
        # Regla simple: si T sube y O2 sube más de 10% respecto a la lectura previa -> inconsistente
        if len(st["hist"]["T"]) >= 2 and len(st["hist"]["O2"]) >= 2:
            if T > st["hist"]["T"][-2] and OD > (st["hist"]["O2"][-2] * 1.10):
                errores.add(10)  # incoherencia T/OD

        # --- Sin error ---
        if not errores:
            errores.add(0)
        return errores

# ==============================
# FILTRO DE PRIORIDAD DE ERRORES
# ==============================
def filtrar_errores_prioritarios(errores):
    """
    Devuelve un subconjunto de errores activos, priorizando los más críticos.
    """
    if not errores: # ningún error activo
        return {0}

    # --- Si hay error serial o logger, ignora todos los de sensores ---
    if {13, 14, 15, 16} & errores:              # Obtenemos la intersección entre ambos conjuntos
(errores críticos ,errores activos)
        return {min({13, 14, 15, 16} & errores)} # se devuelve el error de mayor prioridad (el numero más
bajo de prioridad)

    # --- Si hay error DS18B20, mantiene solo ese y los de comunicación ---
    '''
        -Si hay un error de DS18B20 mientras que hay errores secundarios se conserva
        con ellos en el conjunto de errores activos.
    '''
    if 17 in errores:
        return {17} | ({18, 19, 20} & errores)   # Se hace la unión de conjuntos

    # --- Si hay errores de sensores (pH, O₂, T), se mantienen simultáneos ---
    '''
        -Si existen errores simultáneos de sensores, se mantienen,
        excepto en los casos de redundancia (1 y 11) o (2 y 12).
    '''
    sensores = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} & errores # Solo devuelve los errores de sensores

    # --- Si hay conflictos como (1 y 11) o (2 y 12) ---
    ''' -Si existe (1 y 2) y viceversa (mantener)
        -Si existe (11 y 12) y viceversa (mantener)
        -Si existe (11 y 1) juntos, eliminar 1 (redundancia)
        -Si existe (12 y 2) juntos, eliminar 2 (redundancia)
        -Si existe (1 y 12) o (2 y 11) juntos, mantener ambos (no hay redundancia)
        -Si existe (10 y 12) juntos, eliminar 10 (redundancia)
    '''
    if 11 in sensores and 1 in sensores:  # Si hay ambos errores de pH se descarta el error 1
        sensores.discard(1)
    if 12 in sensores and 2 in sensores:  # Si hay ambos errores de O2 se descarta el error 2
```

```
        sensores.discard(2)
    if 12 in sensores and 10 in sensores: # Si hay incoherencia T/O2 y fluctuación anómala O2, descarta incoherencia
        sensores.discard(10)

    # --- Mantener errores de comunicación secundarios ---
    '''
        -Si hay errores secundarios de forma simultanea se mantienen en
        el conjunto de errores activos.
    '''
    secundarios = {18, 19, 20} & errores  # Si hay simultáneamente errores de comunicación se mantienen

    return sensores | secundarios

# ==============================
# BUCLE PRINCIPAL
# ==============================
acumulador = {canal: 0 for canal in CANALES}
contador = 0
voltaje_suavizado = {canal: None for canal in CANALES}
ultima_actividad = time.time()       # Inicializar watchdog interno
try:
    while True:
        t0 = time.time()             # marca temporal inicio de ciclo
        escuchar_confirmacion_tcp()  # Revisa si llegó alguna orden externa de reanudación
        # --- Reconexión preventiva si 'ser' está caído ---
        if ser is None:
            print("[WARN] Serial no disponible. Intentando reconectar...")
            ser = conectar_serial()
            if ser is None:
                time.sleep(2)
                continue  # reintentar sin caer
            else:
                errores_activos.clear()  # puerto reconectado, limpiar errores

        # --- Acceso seguro a in_waiting ---
        try:
            if ser.in_waiting > 150:
                print(f"[WARN] Buffer saturado ({ser.in_waiting} bytes). Reiniciando buffer...")
                ser.reset_input_buffer()
        except (AttributeError, serial.SerialException, OSError) as e:
            # Algo pasó con el puerto; marcamos error y forzamos reconexión
            errores_activos.add(15)
            print(f"[ERROR] Acceso a puerto serial: {e}. Forzando reconexión...")
            # --- Desactivar PIDs ante error crítico de comunicación ---
            pid_ph_up.auto_mode = False
            pid_ph_down.auto_mode = False
            pid_o2.auto_mode = False
            GPIO.output(RELAY_PIN_PH_UP, GPIO.LOW)
            GPIO.output(RELAY_PIN_PH_DOWN, GPIO.LOW)
            GPIO.output(RELAY_PIN_O2, GPIO.LOW)
            GPIO.output(GPIO_TRIGGER, GPIO.HIGH)
            pulso_reset()
            prev_ph_up = False
            prev_ph_down = False
            try:
                ser.close()
```

```
            except:
                pass
            ser = None

            errores_filtrados = filtrar_errores_prioritarios(errores_activos)
            errores_udp_str = "|".join(str(e) for e in sorted(errores_filtrados))
            try:
                enviar_datos_udp(
                    0, 0, 0, 0,    # voltajes y lecturas nulas
                    0, 0, 0,       # controles
                    pid_ph_up, pid_ph_down, pid_o2,
                    25.0,          # temperatura dummy
                    0, 0, 0,       # tiempos de actuación
                    errores_udp_str
                )
            except:
                pass
            time.sleep(2)
            continue  # seguir vivo, reintentar

        # --- Resto de la iteración protegido ---
        try:
            lectura = leer_linea_ultima()
            if lectura is None:
                time.sleep(DELAY_MUESTREO)
                continue

            promedio, listo, acumulador, contador = procesar_lectura(lectura, acumulador, contador)
            if not listo:
                time.sleep(DELAY_MUESTREO)
                continue

            voltaje_a0, voltaje_a1, pH_compensado, OD_compensado, temperatura_float, voltaje_suavizado =
convertir_y_compensar(promedio, voltaje_suavizado)

            monitorear_seguridad_ph(pH_compensado)

            errores_activos = verificar_sensores(pH_compensado, OD_compensado, temperatura_float)
            gestionar_pids_por_error(errores_activos)

            errores_filtrados = filtrar_errores_prioritarios(errores_activos)
            errores_udp_str = "|".join(str(e) for e in sorted(errores_filtrados)) if errores_filtrados
else "0"
            codigo_error_actual = min(errores_filtrados) if errores_filtrados else 0

            control_ph_down, control_ph_up, control_o2 = actualizar_pid(pH_compensado, OD_compensado)

            mostrar_datos_consola(voltaje_a0, voltaje_a1, pH_compensado, OD_compensado,
                                  control_ph_down, control_ph_up, control_o2,
                                  pid_ph_up, pid_ph_down, pid_o2, temperatura_float,
                                  ser, errores_udp_str)

            procesar_tareas_manuales(time.time())

            (inicio_ciclo_ph_up, inicio_ciclo_ph_down, inicio_ciclo_o2,
             tiempo_on_up, tiempo_on_down, tiempo_on_o2) = actualizar_actuadores(
                control_ph_up, control_ph_down, control_o2,
```

```
                inicio_ciclo_ph_up, inicio_ciclo_ph_down, inicio_ciclo_o2
            )

            enviar_datos_udp(voltaje_a0, voltaje_a1, pH_compensado, OD_compensado,
                             control_ph_down, control_ph_up, control_o2,
                             pid_ph_up, pid_ph_down, pid_o2, temperatura_float,
                             tiempo_on_up, tiempo_on_down, tiempo_on_o2,
                             errores_udp_str)


            # ===== ENVIAR ESTADO COMPLETO AL FLASK =====
            estado = {
                "pid_paused_ph": pid_paused_ph,
                "pid_paused_o2": pid_paused_o2,
                "bloqueo_ph": bloqueo_seguridad_ph,
                "bloqueo_ph_tipo": bloqueo_ph_tipo,
                "bloqueo_ph_hasta": bloqueo_ph_hasta,
                "ph_base": getattr(monitorear_seguridad_ph, "ph_base", None),
                "variacion_acumulada": getattr(monitorear_seguridad_ph, "variacion_acumulada", 0.0),
                "ultimo_ph_up": ultimo_ph_up,
                "ultimo_ph_down": ultimo_ph_down,
                "cooldown": COOLDOWN_PH_SEG,
                "tareas": tareas_manual,
               "timestamp": time.time()
            }
            estado_json = json.dumps(estado)
            enviar_estado_udp_flask(estado_json)


            # --- Watchdog interno: reinicio automático si no hay actividad ---
            # Si no se ha completado una iteración normal en más de 20 segundos,
            # se asume que el programa está bloqueado y se reinicia automáticamente.
            if (time.time() - ultima_actividad) > 20:
                print("[WATCHDOG] Sin actividad. Reiniciando programa...")
                os.execv(sys.executable, ['python3'] + sys.argv)

            # Actualizar marca de tiempo de última actividad (cada ciclo exitoso)
            ultima_actividad = time.time()

            # --- Mantener ciclo de muestreo constante ---
            t0 += DELAY_MUESTREO    # siguiente marca temporal objetivo
            dt = t0 - time.time()   # tiempo restante hasta la siguiente iteración
            if dt > 0:              # dormir solo si queda tiempo
                time.sleep(dt)

        except Exception as e:
            # Cualquier error de la iteración NO debe terminar el proceso
            errores_activos.add(20)
            print(f"[ERROR BUCLE - iteración] {e}")
            time.sleep(0.5)
            continue

except KeyboardInterrupt:
    print("\n[SALIDA] Usuario interrumpió ejecución.")
finally:
    try:
        tcp_socket.close()
```

```
        if ser:
            ser.close()
    except:
        pass
    try:
        udp_socket.close()
    except:
        pass
    GPIO.cleanup()
    print("[INFO] Recursos liberados correctamente.")
```

Anexo E.2.2. Código de logger_3.py

```
# ================================================================
# 🐋 LOGGER v3 — Ultra estable con SQLite WAL + detección inicio/fin de errores
# ================================================================

import socket
import time
import csv
import os
import sqlite3
from datetime import datetime

# ==============================
# CONFIGURACIÓN
# ==============================
UDP_IP = "127.0.0.1"
UDP_PORT = 5006
BUFFER_SIZE = 1024
INTERVALO_REGISTRO = 5.0
DIRECTORIO = "registros_csv"
os.makedirs(DIRECTORIO, exist_ok=True)

ARCHIVO_FIJO = os.path.join(DIRECTORIO, "datos_actual.csv")
MAX_LINEAS_FIJO = 600

DB_FILE = "monitoreo.db"

# ==============================
# DICCIONARIO DE ERRORES
# ==============================
ERROR_DESCRIPCIONES = {
    1: "pH fuera de rango.",
    2: "Oxígeno disuelto fuera de rango.",
    3: "Temperatura fuera de rango.",
    4: "Lecturas de pH inválidas consecutivas.",
    5: "Lecturas de O2 inválidas consecutivas.",
    6: "Lecturas de temperatura inválidas consecutivas.",
    7: "Sensor de pH congelado.",
    8: "Sensor de O2 congelado.",
    9: "Temperatura congelada.",
    10: "Incoherencia entre O2 y temperatura.",
    11: "Fluctuación anómala pH.",
    12: "Fluctuación anómala O2.",
    13: "Error registrador.",
    14: "Error abrir serial.",
```

```
    15: "Error lectura serial.",
    16: "Error decodificación.",
    17: "Error DS18B20.",
    18: "Error enviar UDP.",
    19: "Error GPIO.",
    20: "Error general.",
}

# ==============================
# FUNCIONES AUXILIARES
# ==============================
def crear_nombre_archivo():
    fecha = datetime.now().strftime("%Y-%m-%d")
    return os.path.join(DIRECTORIO, f"datos_{fecha}.csv")

def inicializar_archivo(nombre_archivo):
    encabezados = [
        "fecha", "hora",
        "a0", "pH", "a1", "OD",
        "PID_pH_DOWN", "PID_pH_UP", "PID_O2",
        "P_DOWN", "I_DOWN", "D_DOWN",
        "P_UP", "I_UP", "D_UP",
        "P_O2", "I_O2", "D_O2",
        "T", "t_on_down", "t_on_up", "t_on_o2",
        "codigo_error"
    ]

    if not os.path.exists(nombre_archivo):
        with open(nombre_archivo, "w", newline="") as f:
            csv.writer(f).writerow(encabezados)

def contar_lineas(nombre_archivo):
    try:
        with open(nombre_archivo, "r") as f:
            return sum(1 for _ in f)
    except FileNotFoundError:
        return 0

def crear_socket_udp():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.settimeout(2.0)
    sock.bind((UDP_IP, UDP_PORT))
    return sock

# ==============================
# BASE DE DATOS
# ==============================
def inicializar_bd():
    conn = sqlite3.connect(DB_FILE, timeout=0.5)
    cur = conn.cursor()

    # Modo WAL para escritura concurrente
    cur.execute("PRAGMA journal_mode=WAL;")

    cur.execute("""
    CREATE TABLE IF NOT EXISTS lecturas (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
        fecha TEXT,
        hora TEXT,
        ph REAL,
        o2 REAL,
        temp REAL,
        pid_ph_down REAL,
        pid_ph_up REAL,
        pid_o2 REAL,
        codigo_error TEXT
    )
    """)

    cur.execute("""
    CREATE TABLE IF NOT EXISTS errores_log (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        codigo TEXT NOT NULL,
        descripcion TEXT NOT NULL,
        hora_inicio TEXT,
        hora_fin TEXT,
        resuelto INTEGER DEFAULT 0,
        notificado_inicio INTEGER DEFAULT 0,
        notificado_fin INTEGER DEFAULT 0
    )
    """)

    cur.execute("""
    CREATE TABLE IF NOT EXISTS notificaciones (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        tipo TEXT NOT NULL,
        mensaje TEXT NOT NULL,
        hora TEXT NOT NULL,
        leida INTEGER DEFAULT 0
    )
    """)

    conn.commit()
    conn.close()

def insertar_lectura(fila):
    try:
        conn = sqlite3.connect(DB_FILE, timeout=0.5)
        cur = conn.cursor()

        fecha, hora = fila[0], fila[1]
        ph = float(fila[3])
        o2 = float(fila[5])
        pid_ph_down = float(fila[6])
        pid_ph_up = float(fila[7])
        pid_o2 = float(fila[8])
        temp = float(fila[18])
        codigo_error = str(fila[-1])

        cur.execute("""
            INSERT INTO lecturas (fecha, hora, ph, o2, temp, pid_ph_down, pid_ph_up, pid_o2, codigo_error)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (fecha, hora, ph, o2, temp, pid_ph_down, pid_ph_up, pid_o2, codigo_error))
```

```
        conn.commit()
        conn.close()

    except Exception as e:
        print("[WARN BD] Error al insertar lectura:", e)

def registrar_cambios_de_error(nuevos, resueltos, ahora):
    ts = ahora.strftime("%Y-%m-%d %H:%M:%S")

    try:
        conn = sqlite3.connect(DB_FILE, timeout=0.5)
        cur = conn.cursor()

        # Nuevos errores
        for c in nuevos:
            desc = ERROR_DESCRIPCIONES.get(c, f"Error desconocido ({c})")
            cur.execute("""
                INSERT INTO errores_log (codigo, descripcion, hora_inicio)
                VALUES (?, ?, ?)
            """, (str(c), desc, ts))

        # Errores resueltos
        for c in resueltos:
            cur.execute("""
                UPDATE errores_log
                SET hora_fin = ?
                WHERE codigo = ? AND hora_fin IS NULL
            """, (ts, str(c)))

        conn.commit()
        conn.close()

    except Exception as e:
        print("[WARN BD] Error registrando cambios de error:", e)

# ==============================
# PROCESO PRINCIPAL
# ==============================
def registrar_datos():
    inicializar_bd()

    nombre_actual = crear_nombre_archivo()
    inicializar_archivo(nombre_actual)
    inicializar_archivo(ARCHIVO_FIJO)

    ultimo_error = set()
    ultimo_registro = 0
    datos_ultima_muestra = None
    sock = None

    while True:

        # Rotación diaria
        nombre_nuevo = crear_nombre_archivo()
        if nombre_nuevo != nombre_actual:
            nombre_actual = nombre_nuevo
            inicializar_archivo(nombre_actual)
```

```
if sock is None:
    try:
        sock = crear_socket_udp()
    except:
        time.sleep(2)
        continue

try:
    data, _ = sock.recvfrom(BUFFER_SIZE)
    partes = data.decode().strip().split(",")

    if len(partes) == 21:
        datos_ultima_muestra = partes

except socket.timeout:
    pass

except Exception:
    try: sock.close()
    except: pass
    sock = None
    continue

if time.time() - ultimo_registro >= INTERVALO_REGISTRO and datos_ultima_muestra:

    ahora = datetime.now()
    fecha = ahora.strftime("%Y-%m-%d")
    hora = ahora.strftime("%H:%M:%S")

    fila = [fecha, hora] + datos_ultima_muestra

    try:
        # CSV diario
        with open(nombre_actual, "a", newline="") as f:
            csv.writer(f).writerow(fila)

        # CSV fijo rotativo
        if contar_lineas(ARCHIVO_FIJO) >= MAX_LINEAS_FIJO:
            inicializar_archivo(ARCHIVO_FIJO)

        with open(ARCHIVO_FIJO, "a", newline="") as f:
            csv.writer(f).writerow(fila)

        # ========================
        # MANEJO DE ERRORES
        # ========================
        codigo_error = fila[-1]
        actual = set()

        if codigo_error != "0":
            actual = {int(x) for x in codigo_error.split("|") if x.isdigit()}

        nuevos = actual - ultimo_error
        resueltos = ultimo_error - actual

        if nuevos or resueltos:
```

```
                    registrar_cambios_de_error(nuevos, resueltos, ahora)

                ultimo_error = actual

                insertar_lectura(fila)

            except Exception as e:
                print("[WARN] Registro fallido:", e)

            ultimo_registro = time.time()

        time.sleep(0.1)

if __name__ == "__main__":
    registrar_datos()
```

Anexo E.2.3. Código de app.py (Flask servidor web)

```
from flask import Flask, jsonify, render_template, request
import sqlite3
import os
import socket
import threading
import json
from datetime import datetime, timedelta
import time
import requests

# ================================
# CONFIG
# ================================
app = Flask(__name__, template_folder="templates", static_folder="static")

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DB_FILE = os.path.join(BASE_DIR, "../PID/monitoreo.db")

# ================================
# CLIMA DE OPEN WEATHER KEY
# ================================
OWM_KEY = "7705c2d938df87040e49e60ac1d3d6d3"
LAT = -25.496960843657554
LON = -56.45972174599361

_clima_cache = {
    "ultimo": None,
    "expira": datetime.min,
}
simul_api_calls = 0
estado_v24 = {} #variable global en flask

# ================================
# HELPERS BD
# ================================
def get_db():
    conn = sqlite3.connect(DB_FILE, timeout=0.5)
    conn.row_factory = sqlite3.Row
    return conn
```

```
def init_wal():
    try:
        conn = sqlite3.connect(DB_FILE)
        conn.execute("PRAGMA journal_mode=WAL;")
        conn.commit()
        conn.close()
    except:
        pass

init_wal()

# ------------------------------
# GUARDAR NOTIFICACIONES (seguro)
# ------------------------------
def push_notificacion(tipo, mensaje):
    """Inserta una notificación sin bloquear si la BD está ocupada."""
    ts = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    for _ in range(8):  # hasta 8 reintentos
        try:
            conn = sqlite3.connect(DB_FILE, timeout=0.3)  # timeout corto
            cur = conn.cursor()
            cur.execute("""
                INSERT INTO notificaciones (tipo, mensaje, hora, leida)
                VALUES (?, ?, ?, 0)
            """, (tipo, mensaje, ts))
            conn.commit()
            conn.close()
            return True
        except sqlite3.OperationalError as e:
            if "locked" in str(e).lower():
                time.sleep(0.12)
                continue
            else:
                print("⚠ Error inesperado en push_notificacion:", e)
                return False

    print("❌ No se pudo insertar notificación (BD ocupada).")
    return False


def safe_update(conn, query, params=()):
    """Ejecuta UPDATE con reintentos si la BD está ocupada."""
    for _ in range(8):
        try:
            cur = conn.cursor()
            cur.execute(query, params)
            conn.commit()
            return True
        except sqlite3.OperationalError as e:
            if "locked" in str(e).lower():
                time.sleep(0.12)
                continue
            raise
    print("❌ UPDATE falló tras reintentos (BD ocupada)")
    return False
```

```
# ==============================
# TCP → V24
# ==============================
def enviar_tcp(comando):
    HOST = "127.0.0.1"
    PORT = 5010

    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(1.0)
        sock.connect((HOST, PORT))
        sock.sendall(comando.encode())
        sock.close()
        print(f"[FLASK → V24] TCP enviado: {comando}")
        return True
    except Exception as e:
        print(f"⚠ Error TCP desde Flask: {e}")
        return False

def hilo_udp_estado_v24():
    global estado_v24
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(("0.0.0.0", 6000))
    print("[FLASK] UDP estado escuchando en 6000...")

    while True:
        data, addr = sock.recvfrom(4096)
        try:
            estado_v24 = json.loads(data.decode())
        except:
            pass

# lanzar hilo
threading.Thread(target=hilo_udp_estado_v24, daemon=True).start()

#Registrar tabla acciones_manual si no existe
def init_tabla_acciones_manual():
    conn = get_db()
    conn.execute("""
        CREATE TABLE IF NOT EXISTS acciones_manual (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            fecha TEXT NOT NULL,
            hora TEXT NOT NULL,
            accion TEXT NOT NULL,
            descripcion TEXT NOT NULL,
            valor TEXT,
            estanque INTEGER DEFAULT 1
        )
    """)
    conn.commit()
    conn.close()

# Llamalo al iniciar la app
init_tabla_acciones_manual()
```

```
# Registrar acción manual
def registrar_accion_manual(accion, descripcion, valor=None, estanque=1):
    ahora = datetime.now()
    fecha = ahora.strftime("%Y-%m-%d")
    hora = ahora.strftime("%H:%M:%S")

    conn = get_db()
    conn.execute("""
        INSERT INTO acciones_manual (fecha, hora, accion, descripcion, valor, estanque)
        VALUES (?, ?, ?, ?, ?, ?)
    """, (fecha, hora, accion, descripcion, valor, estanque))
    conn.commit()
    conn.close()


def interpretar_cmd(cmd):
    """
    Recibe el comando EXACTO enviado desde el frontend.
    Devuelve:
        accion_interna (string para BD)
        descripcion    (texto bonito para historial)
        valor          (por ejemplo '20 mL' o '30 s')
    """

    partes = cmd.split()

    # ======================================================================
    # COMANDOS SIN PARÁMETRO
    # ======================================================================
    if cmd == "1":
        return ("pid_ph_play", "PID de pH reanudado", None)

    if cmd == "2":
        return ("pid_o2_play", "PID de O₂ reanudado", None)

    if cmd == "3":
        return ("pid_ambos_play", "PID de pH y O₂ reanudados", None)

    if cmd == "4":
        return ("pid_ph_pause", "PID de pH pausado (modo manual)", None)

    if cmd == "5":
        return ("pid_o2_pause", "PID de O₂ pausado (modo manual)", None)

    if cmd == "6":
        return ("todos_off", "Todos los actuadores detenidos (modo manual)", None)

    # ======================================================================
    # COMANDOS CON PARÁMETRO
    # ======================================================================

    if len(partes) == 2:
        op, val = partes

        # pH ↑ — cal viva
        if op == "8":
            ml_map = {"0": "10 mL", "1": "20 mL", "2": "40 mL"}
```

```
            ml = ml_map.get(val, val)
            return ("dosificacion_ph_up", f"Dosificación pH↑ ({ml})", ml)

        # pH ↓ — ácido
        if op == "9":
            ml_map = {"0": "10 mL", "1": "20 mL", "2": "40 mL"}
            ml = ml_map.get(val, val)
            return ("dosificacion_ph_down", f"Dosificación pH↓ ({ml})", ml)

        # Aireador tiempo
        if op == "7":
            segundos = int(val)
            return ("aireador_on", f"Aireador encendido por {segundos} s", f"{segundos} s")

        # Extensiones del comando 6
        if op == "6" and val == "1":
            return ("aireador_off", "Aireador apagado manualmente", None)

        if op == "6" and val == "2":
            return ("parada_emergencia", "PARADA DE EMERGENCIA: Todos los actuadores apagados", None)

    # ========================================================================
    # DESCONOCIDO
    # ========================================================================
    return ("desconocido", f"Comando manual desconocido: {cmd}", cmd)


# ==============================
# UI PRINCIPAL
# ==============================
@app.route("/")
def index():
    return render_template("mobile.html")


# ==============================
# API: Última lectura sensores
# ==============================
@app.route("/api/sensores")
def api_sensores():
    conn = get_db()
    row = conn.execute("""
        SELECT ph, o2, temp, codigo_error
        FROM lecturas
        ORDER BY id DESC LIMIT 1
    """).fetchone()
    conn.close()

    if not row:
        return jsonify({"ph": None, "o2": None, "temp": None, "error": 0})

    return jsonify({
        "ph": row["ph"],
        "o2": row["o2"],
        "temp": row["temp"],
        "error": row["codigo_error"]
    })
```

```python
# ================================
# API: CLIMA REAL + CACHÉ OPENWEATHER
# ================================
@app.route("/api/clima")
def api_clima():
    global simul_api_calls, _clima_cache

    ahora = datetime.now()

    # Si el cache sigue válido → devolvemos lo almacenado
    if ahora < _clima_cache["expira"]:
        return jsonify({
            "fuente": "cache",
            "api_calls": simul_api_calls,
            "data": _clima_cache["ultimo"]
        })

    # Intentar llamar a OpenWeather
    try:
        url = (
            f"https://api.openweathermap.org/data/2.5/weather"
            f"?lat={LAT}&lon={LON}&appid={OWM_KEY}&units=metric&lang=es"
        )

        r = requests.get(url, timeout=3)
        weather = r.json()

        # Convertir viento de m/s → km/h
        try:
            if "wind" in weather and "speed" in weather["wind"]:
                weather["wind"]["speed"] = round(weather["wind"]["speed"] * 3.6, 2)
        except:
            pass

        simul_api_calls += 1

        # Guardar en caché por 4 minutos
        _clima_cache["ultimo"] = weather
        _clima_cache["expira"] = ahora + timedelta(minutes=4)

        return jsonify({
            "fuente": "openweather",
            "api_calls": simul_api_calls,
            "data": weather
        })

    except Exception as e:
        print("⚠ ERROR consultando OpenWeather:", e)

        # Si falla: devolver cache previo
        if _clima_cache["ultimo"] is not None:
            return jsonify({
                "fuente": "fallback-cache",
                "api_calls": simul_api_calls,
                "data": _clima_cache["ultimo"]
            })
```

```
        # Sin cache → devolver algo mínimo para no romper el frontend
        return jsonify({
            "fuente": "fallback",
            "api_calls": simul_api_calls,
            "data": {
                "name": "Clima no disponible",
                "main": {"temp": 0, "humidity": 0},
                "wind": {"speed": 0}
            }
        })


# ==============================
# API: Errores pendientes (multiusuario)
# ==============================
@app.route("/api/errores_pendientes")
def api_errores_pendientes():
    conn = get_db()
    cur = conn.cursor()

    # ====== 1. Notificar comienzos de error ======
    nuevos = cur.execute("""
        SELECT id, codigo, descripcion
        FROM errores_log
        WHERE hora_inicio IS NOT NULL AND notificado_inicio = 0
    """).fetchall()

    for r in nuevos:
        ok = push_notificacion(
            "error",
            f"⚠ Error [{r['codigo']}] {r['descripcion']} detectado"
        )
        if ok:
            cur.execute("""
                UPDATE errores_log
                SET notificado_inicio = 1
                WHERE id = ?
            """, (r["id"],))
            conn.commit()

    # ====== 2. Notificar errores resueltos ======
    resueltos = cur.execute("""
        SELECT id, codigo, descripcion
        FROM errores_log
        WHERE hora_fin IS NOT NULL AND notificado_fin = 0
    """).fetchall()

    for r in resueltos:
        ok = push_notificacion(
            "resuelto",
            f"✔ Error [{r['codigo']}] {r['descripcion']} resuelto"
        )
        if ok:
            cur.execute("""
                UPDATE errores_log
                SET notificado_fin = 1
```

```
                WHERE id = ?
            """, (r["id"],))
            conn.commit()

    # ====== 3. Enviar errores pendientes para la UI ======
    rows = cur.execute("""
        SELECT id, codigo, descripcion, hora_inicio, hora_fin
        FROM errores_log
        WHERE resuelto = 0
        ORDER BY hora_inicio ASC
    """).fetchall()

    conn.close()

    if not rows:
        return jsonify({
            "hay_error": False,
            "errores": [],
            "hay_activos": False,
            "todos_resueltos": True,
            "reiniciar_ph": False,
            "reiniciar_o2": False
        })

    errores = []
    hay_activos = False
    rein_ph = False
    rein_o2 = False

    PID_PH = {1, 4, 7, 11}
    PID_O2 = {2, 5, 8, 12}
    PID_TEMP = {3, 6, 9, 10, 17, 20}

    for r in rows:
        codigo = int(r["codigo"])
        errores.append({
            "id": r["id"],
            "codigo": codigo,
            "descripcion": r["descripcion"],
            "hora_inicio": r["hora_inicio"],
            "hora_fin": r["hora_fin"]
        })

        if r["hora_fin"] is None:
            hay_activos = True

        if codigo in PID_PH:
            rein_ph = True
        if codigo in PID_O2:
            rein_o2 = True
        if codigo in PID_TEMP:
            rein_ph = True
            rein_o2 = True

    return jsonify({
        "hay_error": True,
        "errores": errores,
```

```
        "hay_activos": hay_activos,
        "todos_resueltos": not hay_activos,
        "reiniciar_ph": rein_ph,
        "reiniciar_o2": rein_o2
    })


# ================================
# API: Reiniciar PIDs
# ================================
@app.route("/api/reiniciar_pids", methods=["POST"])
def api_reiniciar_pids():
    data = request.json
    rein_ph = bool(data.get("reiniciar_ph", False))
    rein_o2 = bool(data.get("reiniciar_o2", False))

    ok_tcp = True

    if rein_ph and rein_o2:
        ok_tcp &= enviar_tcp("3")
    elif rein_ph:
        ok_tcp &= enviar_tcp("1")
    elif rein_o2:
        ok_tcp &= enviar_tcp("2")

    if not ok_tcp:
        return jsonify({"ok": False, "msg": "tcp_fail"}), 500

    # ====== Actualizar BD ======
    conn = sqlite3.connect(DB_FILE, timeout=0.3)
    ts = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    PID_PH = ('1', '4', '7', '11')
    PID_O2 = ('2', '5', '8', '12')
    PID_TEMP = ('3', '6', '9', '10', '17', '20')

    if rein_ph:
        safe_update(conn, """
            UPDATE errores_log
            SET hora_fin = ?, resuelto = 1
            WHERE codigo IN (?, ?, ?, ?) AND resuelto = 0
        """, (ts, *PID_PH))

    if rein_o2:
        safe_update(conn, """
            UPDATE errores_log
            SET hora_fin = ?, resuelto = 1
            WHERE codigo IN (?, ?, ?, ?) AND resuelto = 0
        """, (ts, *PID_O2))

    if rein_ph or rein_o2:
        safe_update(conn, """
            UPDATE errores_log
            SET hora_fin = ?, resuelto = 1
            WHERE codigo IN (?, ?, ?, ?, ?, ?) AND resuelto = 0
        """, (ts, *PID_TEMP))
```

```
    conn.close()

    # ====== Notificación de reinicio ======
    if rein_ph and rein_o2:
        push_notificacion("reinicio", "🔄 Reinicio de PID pH y O₂ ejecutado")
    elif rein_ph:
        push_notificacion("reinicio", "🔄 PID de pH reiniciado")
    elif rein_o2:
        push_notificacion("reinicio", "🔄 PID de O₂ reiniciado")

    return jsonify({"ok": True})


# ====================================
# API: Enviar comandos TCP → v24 (control manual)
# ====================================
@app.route("/api/tcp_send", methods=["POST"])
def api_tcp_send():
    data = request.json
    cmd = data.get("cmd", "").strip()

    if not cmd:
        return jsonify({"ok": False, "msg": "comando vacío"}), 400

    # ======================================================
    #   1) INTERPRETAR COMANDO Y REGISTRAR ACCIÓN MANUAL
    # ======================================================
    try:
        accion_interna, descripcion, valor = interpretar_cmd(cmd)

        registrar_accion_manual(
            accion=accion_interna,
            descripcion=descripcion,
            valor=valor,
            estanque=1   # fijo por ahora
        )

    except Exception as e:
        print("❌ Error registrando acción manual:", e)
        # pero dejamos continuar, no queremos romper el comando TCP

    # ======================================================
    #   2) ENVIAR COMANDO REALMENTE POR TCP
    # ======================================================
    ok = enviar_tcp(cmd)

    # ======================================================
    #   3) RESPUESTA AL FRONTEND
    # ======================================================
    return jsonify({"ok": ok})


# ====================================
# API: RECIBIR ESTADO DE V24
# ====================================
@app.route("/api/estado_v24")
def api_estado_v24():
```

```
    return jsonify(estado_v24)

# ==============================
# API: Historial lecturas
# ==============================
@app.route("/api/historial")
def api_historial():
    conn = get_db()
    rows = conn.execute("""
        SELECT fecha, hora, ph, o2, temp, codigo_error
        FROM lecturas
        ORDER BY id DESC LIMIT 200
    """).fetchall()
    conn.close()

    data = [{
        "fecha": r["fecha"],
        "hora": r["hora"],
        "ph": r["ph"],
        "o2": r["o2"],
        "temp": r["temp"],
        "error": r["codigo_error"]
    } for r in rows]

    return jsonify(data)


@app.route("/api/historial_filtros")
def api_historial_filtros():
    tipo = request.args.get("tipo", "lecturas")       # lecturas | manuales | notificaciones
    periodo = request.args.get("periodo", "dia")      # dia | mes | año | rango
    modo = request.args.get("modo", "tabla")          # tabla | grafica

    fecha = request.args.get("fecha")                 # YYYY-MM-DD
    mes = request.args.get("mes")                     # YYYY-MM
    año = request.args.get("año")                     # YYYY
    desde = request.args.get("desde")                 # YYYY-MM-DD
    hasta = request.args.get("hasta")                 # YYYY-MM-DD

    conn = get_db()
    conn.row_factory = sqlite3.Row
    cur = conn.cursor()

    # ======================================================
    #   CONSTRUCCIÓN DEL FILTRO BASE (para lecturas/manuales)
    # ======================================================
    date_filter = None
    params = []

    if periodo == "dia" and fecha:
        date_filter = "fecha = ?"
        params.append(fecha)

    elif periodo == "mes" and mes:
        date_filter = "fecha LIKE ?"
        params.append(f"{mes}-%")
```

```
    elif periodo == "año" and año:
        date_filter = "fecha LIKE ?"
        params.append(f"{año}-%")

    elif periodo == "rango" and desde and hasta:
        date_filter = "fecha BETWEEN ? AND ?"
        params.extend([desde, hasta])

    # ======================================================
    #   MODO: GRÁFICA → SOLO LECTURAS + AGRUPACIONES
    # ======================================================
    if modo == "grafica":

        # No graficamos manuales ni notificaciones → vacío
        if tipo != "lecturas":
            conn.close()
            return jsonify([])

        # -------------------------
        # 1) Lecturas crudas
        # -------------------------
        q = """
            SELECT fecha, hora, ph, o2, temp
            FROM lecturas
        """
        if date_filter:
            q += f" WHERE {date_filter}"
        q += " ORDER BY fecha ASC, hora ASC"

        rows = cur.execute(q, params).fetchall()

        if not rows:
            conn.close()
            return jsonify([])

        lecturas = [
            {
                "fecha": r["fecha"],
                "hora": r["hora"],
                "ph": float(r["ph"]),
                "o2": float(r["o2"]),
                "temp": float(r["temp"]),
            }
            for r in rows
        ]

        # -------------------------
        # 2) Helpers de agrupación
        # -------------------------
        from datetime import datetime as dt

        def k_hora(f, h):
            return f"{f} {h[:2]}:00"

        def k_dia(f, h):
            return f
```

```
        def k_mes(f, h):
            return f[:7]

        def k_anio(f, h):
            return f[:4]

        def agrupar(fn):
            grupos = {}

            for r in lecturas:
                key = fn(r["fecha"], r["hora"])
                g = grupos.setdefault(key, {"sum_ph": 0, "sum_o2": 0, "sum_temp": 0, "n": 0})
                g["sum_ph"] += r["ph"]
                g["sum_o2"] += r["o2"]
                g["sum_temp"] += r["temp"]
                g["n"] += 1

            res = []
            for key in sorted(grupos.keys()):
                g = grupos[key]
                res.append({
                    "categoria": "lectura",
                    "fecha": key.split(" ")[0],
                    "hora": key.split(" ")[1] if " " in key else "",
                    "ph": round(g["sum_ph"] / g["n"], 2),
                    "o2": round(g["sum_o2"] / g["n"], 2),
                    "temp": round(g["sum_temp"] / g["n"], 2),
                    "label": key,
                })

            return res

        # -------------------------
        #     AGRUPACIONES
        # -------------------------
        if periodo == "dia":
            # → promedio por hora
            data = agrupar(k_hora)

        elif periodo == "mes":
            # → promedio por día
            data = agrupar(k_dia)

        elif periodo == "año":
            # → promedio por mes
            data = agrupar(k_mes)

        elif periodo == "rango" and desde and hasta:
            # duración del rango
            try:
                d1 = dt.strptime(desde, "%Y-%m-%d")
                d2 = dt.strptime(hasta, "%Y-%m-%d")
                ndias = (d2 - d1).days + 1
            except:
                ndias = 9999

            if ndias <= 31:
```

```
            data = agrupar(k_dia)
        elif ndias <= 365 * 3:
            data = agrupar(k_mes)
        else:
            data = agrupar(k_anio)
    else:
        data = agrupar(k_dia)

    conn.close()
    return jsonify(data)

# ====================================================
#   MODO TABLA — CONSULTAS INDEPENDIENTES
# ====================================================
consultas = []  # cada elemento será: (query, params)

# ---------------------------
# LECTURAS
# ---------------------------
if tipo == "lecturas":
    q = """
        SELECT fecha, hora, 'lectura' AS categoria,
               ph, o2, temp, codigo_error, 1 AS estanque
        FROM lecturas
    """
    if date_filter:
        q += f" WHERE {date_filter}"
        consultas.append((q, params.copy()))
    else:
        consultas.append((q + " ORDER BY fecha DESC, hora DESC LIMIT 500", []))

# ---------------------------
# ACCIONES MANUALES
# ---------------------------
if tipo == "manuales":
    q = """
        SELECT fecha, hora, 'manual' AS categoria,
               descripcion, valor, accion, estanque
        FROM acciones_manual
    """
    if date_filter:
        q += f" WHERE {date_filter}"
        consultas.append((q, params.copy()))
    else:
        consultas.append((q + " ORDER BY fecha DESC, hora DESC LIMIT 500", []))

# ---------------------------
# NOTIFICACIONES (especial)
# ---------------------------
if tipo == "notificaciones":
    # NOTIFICACIONES NECESITAN FILTRO PROPIO (hora tiene datetime completo)
    q = """
        SELECT
            substr(hora, 1, 10) AS fecha,
            substr(hora, 12) AS hora,
            'notificacion' AS categoria,
            mensaje,
```

```
                tipo,
                leida,
                1 AS estanque
            FROM notificaciones
        """

        # FILTRO ESPECIAL PARA NOTIFICACIONES
        if periodo == "dia" and fecha:
            q += " WHERE substr(hora, 1, 10) = ?"
            consultas.append((q, [fecha]))

        elif periodo == "mes" and mes:
            q += " WHERE substr(hora, 1, 7) = ?"
            consultas.append((q, [mes]))

        elif periodo == "año" and año:
            q += " WHERE substr(hora, 1, 4) = ?"
            consultas.append((q, [año]))

        elif periodo == "rango" and desde and hasta:
            q += " WHERE substr(hora, 1, 10) BETWEEN ? AND ?"
            consultas.append((q, [desde, hasta]))

        else:
            consultas.append((q, []))

    # ====================================================
    #   PAGINACIÓN PARA TABLA
    # ====================================================
    page = int(request.args.get("page", 1))
    limit = int(request.args.get("limit", 30))
    offset = (page - 1) * limit

    # Obtener datos completos en memoria
    final_data = []
    for q, p in consultas:
        final_data.extend([dict(r) for r in cur.execute(q, p).fetchall()])

    # Ordenar todo por fecha/hora descendente
    def sort_key(x):
        f = x.get("fecha", "")
        h = x.get("hora", "")
        return (f, h)

    final_data.sort(key=sort_key, reverse=True)

    # Total de filas y páginas
    total_items = len(final_data)
    total_pages = max(1, (total_items + limit - 1) // limit)

    # Cortar solo la página actual
    page_data = final_data[offset : offset + limit]

    conn.close()

    # Devolver datos con información de paginación
    return jsonify({
```

```
        "page": page,
        "pages": total_pages,
        "limit": limit,
        "total": total_items,
        "data": page_data
    })

# ==============================
# API: Notificaciones
# ==============================
@app.route("/api/notificaciones")
def api_notificaciones():
    conn = get_db()
    rows = conn.execute("""
        SELECT id, tipo, mensaje, hora, leida
        FROM notificaciones
        ORDER BY id DESC
        LIMIT 20
    """).fetchall()
    conn.close()

    return jsonify([
        {
            "id": r["id"],
            "tipo": r["tipo"],
            "mensaje": r["mensaje"],
            "hora": r["hora"],
            "leida": r["leida"]
        }
        for r in rows
    ])


@app.route("/api/notificaciones_no_leidas")
def api_notificaciones_no_leidas():
    conn = get_db()
    row = conn.execute("""
        SELECT COUNT(*) AS cnt
        FROM notificaciones
        WHERE leida = 0
    """).fetchone()
    conn.close()

    return jsonify({"pendientes": row["cnt"]})


@app.route("/api/notificaciones_marcar_leidas", methods=["POST"])
def api_notificaciones_marcar_leidas():
    conn = get_db()
    conn.execute("UPDATE notificaciones SET leida = 1 WHERE leida = 0")
    conn.commit()
    conn.close()
    return jsonify({"ok": True})

@app.route("/api/notificacion_sistema", methods=["POST"])
def api_notificacion_sistema():
    data = request.json
```

```
    tipo = data.get("tipo", "sistema")
    mensaje = data.get("mensaje", "Evento del sistema")

    push_notificacion(tipo, mensaje)
    return jsonify({"ok": True})
```

Anexo F. Registro fotográfico de la interfaz web

Anexo F.1. Menú principal y navegación

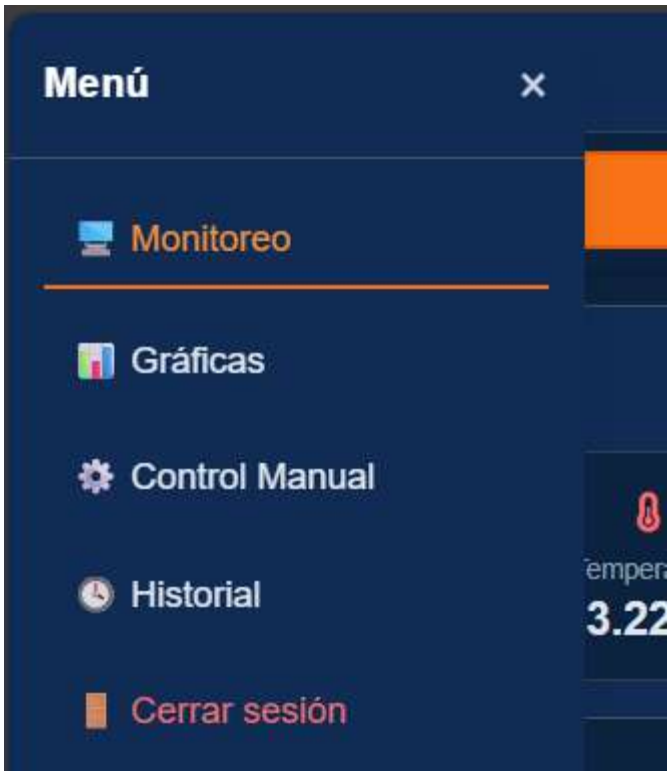

Figura 5.8. Menú lateral de navegación del sistema

Anexo F.2. Panel de monitoreo en tiempo real

Figura 5.9. Visualización en tiempo real de los parámetros del estanque y del clima

Anexo F.3. Gráfica en tiempo real de parámetros

Figura 5.10. Gráfica en tiempo real del sistema

Anexo F.4. Gestión de errores y notificaciones

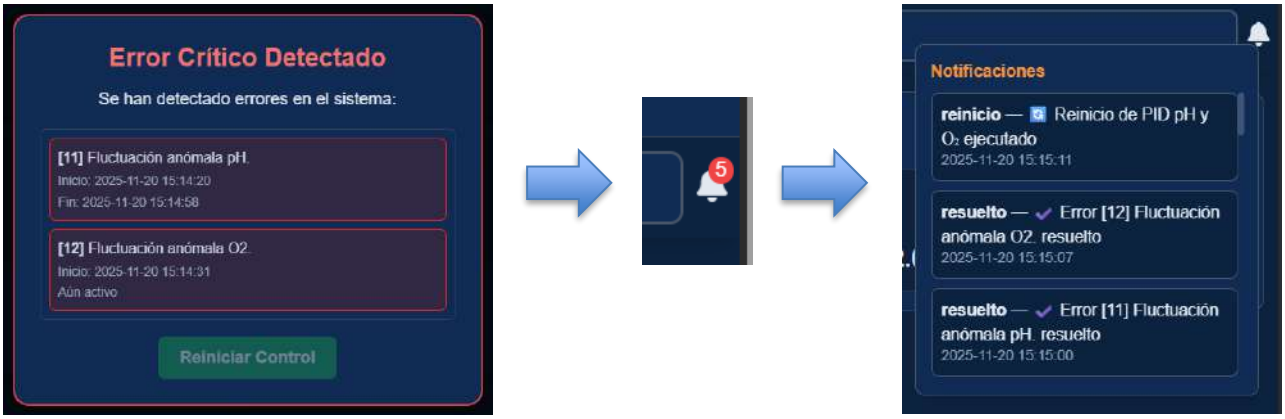

Figura 5.11. Flujo del sistema de notificaciones y alertas

Anexo F.5. Módulo de control manual

Figura 5.12. Interfaz del módulo de control manual del sistema

Anexo F.6. Historial de lecturas y acciones

Figura 5.13. Interfaz del módulo de historial, mostrando filtros por tipo de registro, período, fecha y estanque

Anexo G. Desglose presupuestario

Anexo G.1. Costo general

N°	Componente	Modelo/	Cantidad	Precio unitario	SUBTOTAL
1	Controlador principal	Raspberry Pi 4	1	68	68
2	Microcontrolador secundario	Arduino UNO R3	1	14	14
3	Sensor de Oxígeno Disuelto	SEN0237-A	1	180	180
4	Sensor de pH	GAOHOU PH 0-14	1	29	29
5	Sensor de turbidez	SEN0189	1	9,90	9,90
6	Sensor de temperatura	DS18B20	1	9	9
7	Sensor de temperatura y humedad	DTH11	1	8	8
8	Bombas peristálticas	Kamoer NKP	2	8	16
9	Bloque de cargador	Xiaomi 67W	1	20	20
10	Ficha toma corriete	Kalop	1	0,5	0,5
11	Cable blindado	Inpacont flex blindado	7	20	20
12	UPS raspberry	MakerFocus v3 plus	1	28	28
13	Flotador	Kyo 90764-cd02bu	1	3,8	3,8
14	Potenciómetro	10kΩ	2	1,43	2,86
15	Capacitor electrolítico	220µF	1	0,5	0,5
16	Resistencias	120k, 4.7k, 220	6	0,07	0,07

17	Compuertas OR	74LS32P	1	0,35	0,35
18	Temporizador	555	1	0.7	0.7
19	Conversor ADC	ADS1115 16 bits	1	6,4	6,4
20	Jumpers	Dupont Wire Kit	1	3,4	3,4
21	Caseta	De madera	1	9	9
22	Placa perforada	Genérico	1	2	2
23	Leds	Genérico	4	0,02	0,08
24	Solución para almacenamiento de sensor de OD	KCI-3M	1	6	6
25	Solución para almacenamiento de sensor de pH	KCL-3M	1	6	6
26	Buffers calibración pH	Apera Instruments	1	2,1	2,1
27	Caja distribución	CONATEL 300x250x120	1	11	11
28	Prensaestopas de cable	FZ0083	8	0,35	2,8
29	Manguera	Genérico	10	1,43	14,3
30	Tarjeta de memoria	32 GB sandisk	1	5	5
31	Ácido clorhídrico	HCL 37%	1	32	32
32	Aumentador de alcalinidad	Pool Mate	1	18	18
SUBTOTAL					**$ 528.76**

Tabla 5.2. Costo total de componentes electrónicos

Anexo G.2. Costo total de programación y mano de obra

Concepto	Detalles / Justificación	Costo (USD)
Creación y programación de software	Codificación en Arduino C++, Python y Flask para la creación de la interfaz web	3200
Incorporación de sensores	Ajuste e instalación de sensores de pH, oxígeno disuelto y turbidez	500
Diseño y codificación de la interfaz web	Creación de una interfaz gráfica adaptable y optimizada	150
Trabajo especializado en electrónica	Montaje de componentes, incluyendo soldadura y conexionado	200
Servicio de mantenimiento y asistencia durante 3 meses	Configuración del firmware, calibración de sensores y realización de pruebas constantes	70
SUBTOTAL		$ 4120

Tabla 5.3. Costo total de programación y mano de obra

Anexo H. Registro Fotográfico de la Instalación del Sistema

Anexo H.1. Instalación de mangueras con los recipientes para las soluciones correctivas dentro de un ducto de ladrillos

Figura 5.14 Botellas conectadas a las bombas peristálticas, empleadas para la dosificación de correctivos de pH y ácido

Anexo H.2. Boya flotante instalada en el estanque con los sensores de monitoreo

Figura 5.15 Boya flotante instalada en el estanque, con los sensores de medición alojados en el compartimento interno.

Anexo H.3. Visualización del sitio web en dispositivo móvil

Figura 5.16. Lectura en tiempo real de los parametros del agua

Anexo H.4. Caseta de control del sistema.

Figura 5.17. Vista de la caseta de madera que protege los equipos electrónicos y la Raspberry Pi

Anexo H.5. Interior de la caseta - electrónica del sistema.

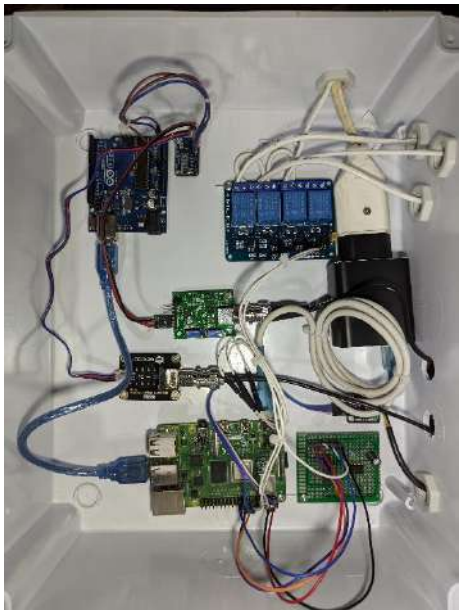

Figura 5.19. Distribución interna de la Raspberry Pi, Arduino, relés, ADS1115 y conexiones eléctricas

Anexo H.6. Bombas peristálticas y aireador

Figura 5.18. Bombas peristálticas conectadas a las mangueras de dosificación, con descarga cercana al aireador para favorecer la recirculación de la solución

Anexo H.7. Vista panorámica

Figura 5.20. Vista completa del estanque y la caseta con la instalación final