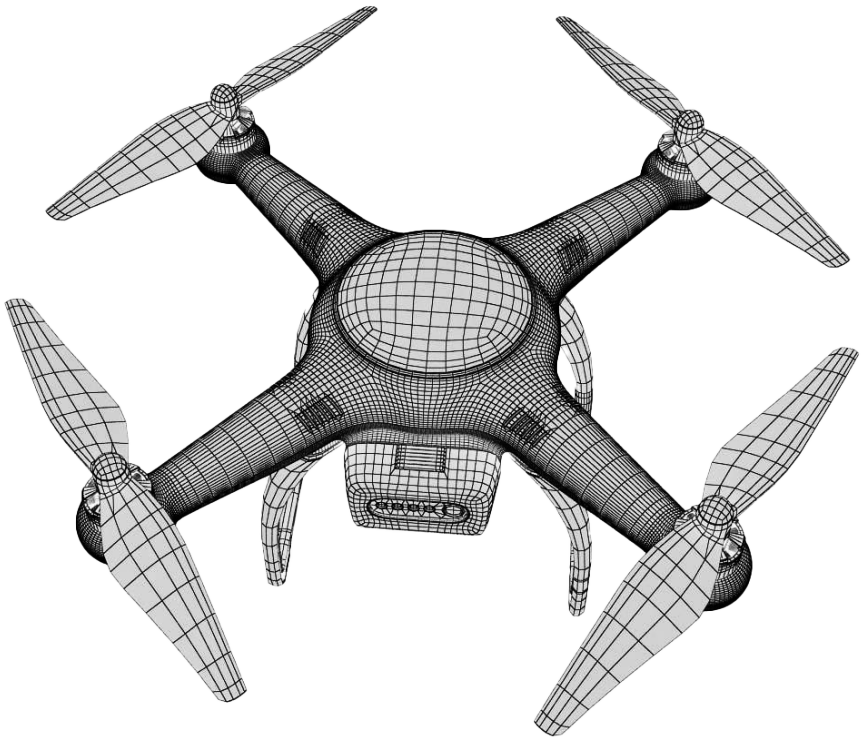
Control Theory Applied to Quadcopters



Víctor Villarejo Alier

Julio César Vera Hernández

La Miranda

December 12, 2025

Cover image by Frezzy on Free3D.

The following QR code provides a link to a GitHub repository containing all code and experimental data used, as well as the finalized research paper in digital format.

In the digital version, this QR code can be directly clicked to be redirected to the repository.

Acknowledgements

I would like to express my sincere gratitude to my supervisor for their invaluable guidance, patience, and support throughout the development of this research project. Their insights were instrumental in navigating the complexities of control theory and shaping the overall direction of this work.

I am also deeply grateful to my father for his essential assistance during the physical experimentation phase. His hands-on help in constructing the experimental rigs and his support in troubleshooting the hardware challenges were fundamental to the practical realization of this project.

Abstract

Unmanned aerial vehicles are inherently unstable systems that rely on continuous feedback and control theory to maintain autonomous flight. This research investigates the application of control strategies to quadcopter stabilization, with the objective of connecting mathematical modeling with practical implementation. The study begins with a comprehensive theoretical framework covering multirotor dynamics and sensor technologies. Additionally, it describes the concepts behind On-Off, Proportional, and Proportional-Integral-Derivative (PID) control algorithms.

To validate these concepts, two experimental rigs, a rotational oscillator and a vertical oscillator, were engineered using an ESP32 microcontroller and brushless motors. However, physical testing was not successful due to electromagnetic interference generated by the high-current motors disrupting I^2C sensor communication, preventing real-time feedback and therefore control. To overcome these setbacks and visualize theoretical behaviors, a numerical simulation was developed in Python. The simulation results showed that, while On-Off control leads to marginal stability and Proportional control results in steady-state error, PID control provides superior stabilization and minimal settling time. Despite hardware integration challenges, the combination of theoretical study, prototyping, and simulation demonstrates the mechanics of drone flight and the necessity of robust, stable, and reliable design in autonomous systems.

Resum

Els vehicles aeris no tripulats són sistemes inherentment inestables que es basen en la retroalimentació contínua i la teoria de control per mantenir el vol autònom. Aquesta investigació es centra en l'aplicació d'estratègies de control a l'estabilització de quadcoptres, amb l'objectiu de connectar la modelització matemàtica amb la implementació pràctica. L'estudi comença amb un marc teòric complet que cobreix la dinàmica dels multirotors i les tecnologies de sensors. A més, descriu els conceptes darrere dels algorismes de control tot-o-res (On-Off), proporcional i proporcional-integral-derivatiu (PID).

Per validar aquests conceptes, es van dissenyar dos equips experimentals, un oscil · lador rotacional i un oscil · lador vertical, utilitzant un microcontrolador ESP32 i motors sense pinzells. No obstant això, les proves físiques no van tenir èxit a causa de la interferència electromagnètica generada pels motors d'alta corrent que van interrompre la comunicació del sensor per I^2C, impedint la retroalimentació en temps real i per tant el control. Per superar aquests contratemps i visualitzar els

comportaments teòrics, es va desenvolupar una simulació numèrica en Python. Els resultats de la simulació van mostrar que, mentre que el control tot-o-res condueix a l'estabilitat marginal i el control proporcional dona lloc a un error d'estat constant, el control PID proporciona una estabilització superior i un temps mínim d'ajustament. Malgrat els reptes d'integració dels components, la combinació d'estudi teòric, prototipatge i simulació demostra la mecànica del vol amb drons i la necessitat d'un disseny robust, estable i fiable en sistemes autònoms.

Resumen

Los vehículos aéreos no tripulados son sistemas inherentemente inestables que dependen de la retroalimentación continua y la teoría de control para mantener el vuelo autónomo. Esta investigación estudia la aplicación de estrategias de control a la estabilización de cuadricópteros, con el objetivo de conectar el modelado matemático con la implementación práctica. El estudio comienza con un marco teórico completo que abarca la dinámica multirrotor y las tecnologías de sensores. Además, describe los conceptos tras los algoritmos de control todo-o-nada (On-Off), proporcional y proporcional-integral-derivativo (PID).

Para validar estos conceptos, se diseñaron dos monturas experimentales, un oscilador rotacional y un oscilador vertical, utilizando un microcontrolador ESP32 y motores sin escobillas. Sin embargo, las pruebas físicas no tuvieron éxito debido a las interferencias electromagnéticas generadas por los motores de alta corriente, que interrumpían la comunicación del sensor por I^2C, lo que impedía la retroalimentación en tiempo real y, por lo tanto, el control. Para superar estos contratiempos y visualizar los comportamientos teóricos, se desarrolló una simulación numérica en Python. Los resultados de la simulación mostraron que, mientras que el control todo-o-nada lleva a una estabilidad marginal y el control proporcional da lugar a un error de estado estacionario, el control PID proporciona una estabilización superior y un tiempo de asentamiento mínimo. A pesar de los retos que planteó la integración de los componentes, la combinación del estudio teórico, el prototipado y la simulación demuestran la mecánica del vuelo de los drones y la necesidad de un diseño robusto, estable y fiable en los sistemas autónomos.

CONTENTS

1 INTRODUCTION

Unmanned aerial vehicles, commonly known as drones, have been rapidly expanding in both scope and scale of use. Their adoption spans a wide range of applications, from search-and-rescue operations to agricultural monitoring, infrastructure inspection, or even commercial delivery systems. As these devices are expected to operate autonomously and at scale, it is a critical challenge to ensure their safety, reliability, and stability. An autonomous drone navigating through an agricultural field, an industrial site, or an urban environment must rely on robust internal systems to interpret its environment, maintain its orientation, and react safely to disturbances. The failure of such a mechanism can have consequences ranging from loss of equipment to physical harm on bystanders. For these reasons, I believe it is crucial to understand how physical systems are controlled and stabilized to be able to design safe and effective autonomous aerial systems.

This paper aims to study how physical systems can be regulated through closed-loop control, both mathematically and in practical implementations, using drones as an example of a system to be stabilized. Although drones are the main context of the paper, the central theme is control theory itself, which is the framework with which dynamical systems can be modeled, stabilized, and adapted to unpredictable conditions. Drones are an ideal case study for this, due to their unstable nature (unlike other aerial systems, such as planes, which can glide, drones will drop when not controlled) and their reliance on continuous feedback to maintain position, altitude, and orientation.

My personal motivation for this project comes from my long-standing interest in drones, especially first-person-view (FPV) flight. Flying quadcopters in simulations fascinated me not only for their maneuverability but also for their perceived sense of freedom, being able to navigate in any direction and altitude. Another factor that contributed to my appreciation for these systems was the sophisticated stabilization systems within commercial drones. The fact that a device such as DJI's recent drones can remain still in mid-air against heavy wind or other adverse circumstances motivated me to understand the logic behind such performances. Through this work, I have been able to convert this curiosity into hands-on experience, learning from both the successes of the project and its setbacks.

The objectives of this paper reflect this interest in both theory and practice. First, my aim is to understand the fundamental components of control systems by studying how drones utilize sensors, actuators, and feedback to maintain stability. Second, I seek to learn how classical control strategies, such as On-Off, Proportional, and PID control, are implemented, and how their theoretical properties manifest when applied. A third objective is to build simplified models of drone subsystems to observe their behavior in controlled conditions and

analyze how different control strategies affect their stability. Fourth, I intend to observe these strategies' limitations when implemented in real systems, including sensor imperfections, noise, and other constraints that do not appear in idealized theoretical models. Finally, this project aims to connect the mathematical descriptions of control with their real-world behavior.

To address these research objectives, I have adopted an experimental methodology supported by both physical prototypes and a virtual simulation. Two experimental rigs were built to model aspects of drone dynamics: a rotational oscillator to emulate pitch and roll stabilization, and a vertical oscillator to represent hover control along the vertical axis. Each rig incorporates the types of sensors commonly found in drones, such as gyroscopes, accelerometers, barometric altimeters, and time-of-flight distance sensors, combined with brushless motors akin to those found in real drones and a microcontroller to handle the control algorithms in real-time. To complement these physical experiments, a Python simulation has been developed to explore the consequences of different control strategies in a semi-idealized environment. This simulation allows for the analysis of stability, steady-state error, transient response, and other metrics that will be explained in the following sections.

2 THEORETICAL FRAMEWORK

This theoretical framework will cover all aspects of quadcopters necessary to understand the practical and virtual experiments carried out in Section 3, from the sensors that they rely on to navigate, to the logic that they employ to decide their best course of action to achieve their goal.

2.1 DRONE BASICS

A quadcopter, colloquially called a drone, is a style of multirotor aircraft characterised by four propellers in a vertical position.

As stated before, most drones feature four propellers set out in a cross shape. Propeller orientations (the direction in which the propellers are pitched towards) are shared between diagonally adjacent propellers, as shown in Figure 2.1, to counteract the torque effect. The torque effect is the resulting force on the drone due to Newton's third law, where the propellers create reverse rotational motion on the aircraft itself. In helicopters, this issue is mitigated through using a tail rotor or having coaxial rotors.

In order to operate, drones have four basic movements which can be combined to attain any desired position and orientation. These four movements are thrust, roll, pitch, and yaw. Their respective axes are shown in Figure 2.2.

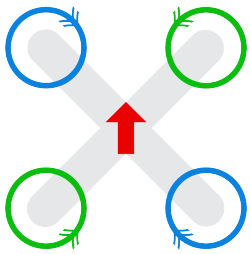


Figure 2.1: Propeller rotation orientations on a typical quadcopter (Own Illustration).

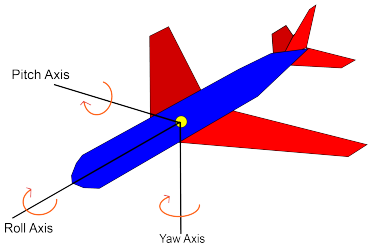


Figure 2.2: Principal rotational axes (Wikipedia Contributors, 2025).

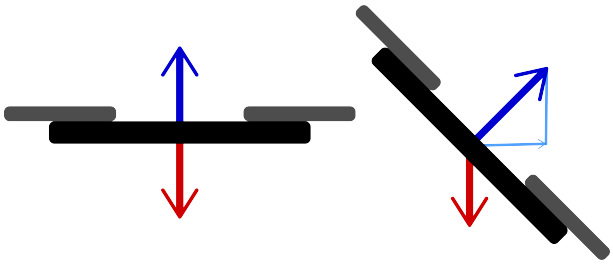


Figure 2.3: Thrust (blue) opposing gravity (red) in two different orientations (Own Illustration).

Thrust refers to the increase in motor power across all four motors, causing the propellers to spin faster and therefore generate lift. This is the only primary movement that involves movement in space, as the other movements only result in a stationary rotation.

Roll involves increasing output power to either both left or right motors, achieving a lateral tilting motion. Similarly, pitch increases motor power to either the front or back propellers, resulting in a forward or backward tilting motion.

Lastly, yaw refers to a rotational movement along the vertical axis. It is achieved by increasing the motor power of diagonally adjacent rotors. This reintroduces the previously mentioned torque effect, resulting in a lateral panning motion akin to shaking one's head (ArduPilot Dev Team, 2025).

By combining these movements, any desired position and orientation can be achieved. For instance, forward movement can be achieved through a light pitch forward followed by thrust, as depicted in Figure 2.3, on the left-hand side.

These principles of quadcopter movement can be further understood through this simplified two-dimensional free-body diagram, which can easily be extended into three dimensions. On the left-hand side of Figure 2.3, thrust (in blue) is shown as a force opposing weight (in red). In this orientation, increasing or decreasing thrust would result in a purely vertical movement. However, if roll is applied, as shown on the right-hand side, the new horizontal component of the thrust force would result in a sideways motion. It is also worth noting that, in this situation, more thrust would be necessary in order to maintain the drone's vertical position.

Quadcopters are unstable systems, which means that, due to their aerodynamics, they will not be able to fly without additional compensation for external factors. Automatic stabilisation is the process through which, by combining primary movements, the quadcopter maintains a stable position, hovering in place. This feature eliminates the need for a pilot to constantly correct the aircraft's position and is managed by the craft's flight controller, which relies on sensor data and various stabilisation algorithms to avoid drift and maintain the drone's position accurately (ArduPilot Dev Team, 2025).

2.2 SENSORS

To achieve autonomous flight and maintain stability, a quadcopter must constantly be aware of its own state within three-dimensional space. As covered in the previous section, multirotor aircraft are inherently unstable systems that require continuous correction to remain airborne. As such, the flight controller relies on a continuous stream of data to counteract external disturbances and maintain the drone's position.

Sensors connect the physical world to digital control logic, converting phenomena such as angular velocity, linear acceleration, or atmospheric pressure into measurable signals. The following subsections detail the operating principles of most sensors that are applicable to drones, with an emphasis on MEMS technology due to its compact size and prevalence in modern robotics.

2.2.1 Gyroscopes

Gyroscopes are small devices that detect rotational motion. Nowadays, the most common type of gyroscope is the MEMS gyroscope. MEMS stands for Micro-Electro-Mechanical Systems, which are systems fabricated at a micrometer scale. The components for these systems are often directly carved into silicon using processes such as lithographic etching, which involves exposing the silicon to radiation such as light in order to carve specific patterns into it (MEMS Exchange, 2025). This includes carving springs, masses, and all other components in order to mimic mechanical systems. Nowadays, MEMS sensors are preferred due to their compact form factor and their inexpensive nature (Nedelkovski, 2015).

MEMS Gyroscopes operate on the Coriolis effect, which is a pseudo-force that appears when an object is observed from a non-inertial frame of reference. A clear example of this is launching objects on Earth.

Consider an object launched forwards at a high speed. Newton's law of inertia states that this object will continue advancing in a rectilinear fashion. Even though the object is moving forwards, due to the spin of the Earth, it appears to follow a curved path relative to us. This apparent curved motion is not due to any force acting on the object, but rather due to the rotating frame of reference for the observer. Mathematically, this pseudoforce can be modeled through the following equation:

$$\vec{F_c} = -2m(\vec{\omega} \times \vec{v})$$

Where:

- $\vec{F_c}$ is the Coriolis force,
- m is the mass of the object,

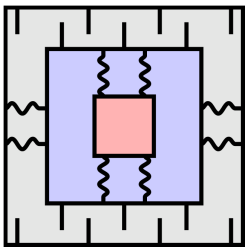


Figure 2.4: Close-up of the sensing structure in a MEMS gyroscope (Own Illustration).

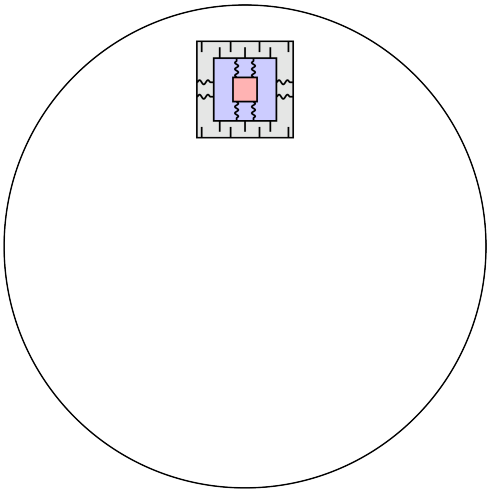


Figure 2.5: Diagram showing the full structural layout of a MEMS gyroscope (Own Illustration).

- $\vec{\omega}$ is the angular velocity vector of the rotating reference frame (for example, the Earth's rotation),
- $\vec{v}$ is the velocity of the object itself.

This effect is what allows MEMS gyroscopes to detect rotation. Within these sensors, there is a microscopic proof mass, which is made to oscillate back and forth along a fixed axis (Analog Devices, 2016). In Figure 2.4, this is the vertical axis.

In this diagram, the proof mass (red) is suspended by vertical springs. The inner frame (blue) is suspended by horizontal springs and contains capacitive sensors.

This sensing structure is then placed onto a disk, as shown in Figure 2.5.

Once assembled, the gyroscope is mounted onto the object whose rotational motion we want to measure. When the object rotates, a Coriolis force is then applied to the sensing structure, which results in the inner frame moving laterally. Lastly, this motion changes the capacitance between the sensing electrodes, allowing angular motion to be measured.

Each sensing structure can detect angular motion in one axis, so most gyroscopes implement three orthogonal structures in order to detect motion over any axis.

It is important to note that, if we observe the Coriolis force equation, there is only a component for angular velocity, and not angular orientation. This means that we will not be able to directly obtain the absolute orientation of the object, but rather only the speed at which it is rotated. To obtain the object's orientation, the angular velocity must be numerically integrated by using Euler's method, for example, which estimates the new orientation based on its rate of change and the time between measurements.

Even though this method can approximate the orientation of the object, it is not perfect. This is because sensors have noise, which is a slight deviation from the correct reading due to external factors. Additionally, numerical integration methods can only approximate results, and they will, by definition, have a slight error in their calculations, which is referred to as drift. Despite this, modern consumer-grade MEMS gyroscopes can achieve orientation error margins under 2° (Passaro et al., 2017).

MEMS gyroscopes are favored due to their compact form factor, their low cost, their robustness, and their accuracy. They are employed in a vast range of applications, such as smartphones, automotive systems, or, more importantly in this context, drones.

2.2.2 Accelerometers

Accelerometers are another useful sensor for flight. Rather than measuring angular movement, they sense linear acceleration. This means that, when a force is applied, the accelerometer can detect its orientation and its magnitude. Just like with gyroscopes, MEMS accelerometers are the most widespread due to their low cost and small size. Since they share the same technology, a lot of terminology and basic operation is shared between these sensors.

MEMS accelerometers work on the principle of linear inertia. According to Newton's laws of motion, an object will resist changes in its state of motion unless it is acted on by an external force. MEMS accelerometers utilize this principle by suspending a proof mass with silicon spring-like structures, as shown in Figure 2.6. When the device accelerates, the mass tends to remain in its original state due to inertia, causing a displacement relative to the rest of the sensor. In Figure 2.6, this would be a lateral movement along the sensing axis. The greater the applied force, the larger the displacement (VectorNav, 2025a).

This displacement can then be detected through capacitive sensing. Just as in MEMS gyroscopes, the proof mass is surrounded by fixed electrodes. When the mass moves, the distance to these electrodes changes, which alters the electric field distribution and therefore the capacitance. This change in capacitance then produces an electrical signal proportional to the applied force (Keim, 2020).

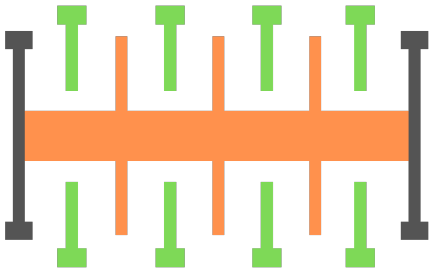


Figure 2.6: Diagram showcasing the components of a MEMS accelerometer. The capacitive sensors are colored green and the proof mass is colored orange, while the springs are colored black (Own Illustration).

Like MEMS gyroscopes, each accelerometer can only measure acceleration along a single axis. Therefore, most commercial accelerometers include three orthogonally aligned units to detect motion in three-dimensional space.

It is also important to note that, since (in simple terms) accelerometers measure the difference in acceleration between the frame and the proof mass, a state of free fall would output a null reading, as both components would be pulled by the Earth at the same rate. Counterintuitively, a stationary accelerometer, such as one on the ground or hovering in midair, would read an upwards acceleration of $9.81\,\mathrm{m\,s^{-2}}$. This occurs because gravity pulls both the frame and the proof mass downward, but the upward force from the surface or the drone's motors acts only on the frame. The proof mass, in contrast, continues to fall freely. Lastly, since the sensor reads the acceleration from the frame relative to the proof mass and not the other way around, the downward pull of the proof mass on the frame is reinterpreted by the sensor as an upward pull of the frame on the mass, resulting in a positive acceleration reading (VectorNav, 2025a).

Another way to interpret this is to think of the accelerometer as if it ignores gravity entirely. Instead, it only detects vertical forces that resist free fall, such as the ground pushing up or a motor lifting the accelerometer. By taking this into account, we are able to determine if the device is accelerating, stationary, or in free fall.

A practical consequence of this behavior is that accelerometers can be used to determine their orientation relative to gravity. When stationary, the only force detected by the sensor is the normal force, which always points upwards, counteracting gravity. Using this fact, we can analyze how the accelerometer's reading is tilted, giving us its pitch and roll angles. As the sensor tilts, the projection of this force onto each axis changes, allowing for the estimation of these angles through vector decomposition (Fisher, 2010).

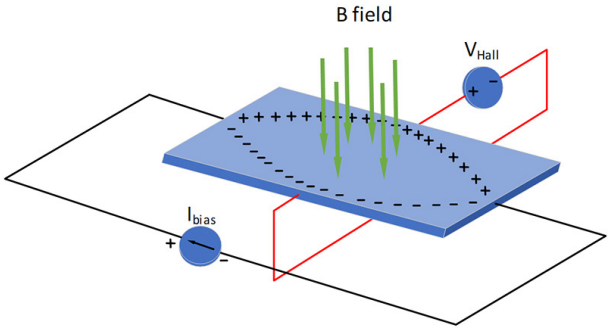


Figure 2.7: Diagram of the Hall effect. The conducting plate is shown in light blue, while the deflection of the charges due to the Lorentz force is represented as $+$ and $-$ trails and the magnetic field B is shown in green. The Hall voltage is measured through the voltmeter connected perpendicular to the current flow (V_{Hall}) (Texas Instruments, 2021).

In drones, accelerometers are not only used for orientation estimation but also for detecting external disturbances. For example, if wind pushes the drone off course, the accelerometer can detect this shift, allowing the control system to apply corrections. Accelerometers are also useful for crash detection. If a large force is suddenly detected but the drone has not increased power to its thrusters, it will assume that a crash has occurred. The drone will then activate safety measures to minimize damage and ensure safety.

2.2.3 Magnetometers

While accelerometers can obtain pitch and roll angles, drones relying solely on them would be missing the third rotational value, yaw. For this, magnetometers are used. Magnetometers are sensors that measure magnetic fields. By measuring the earth's magnetic field, it can determine its heading or yaw. The most common type of magnetometer nowadays is the MEMS Hall-effect magnetometer (VectorNav, 2025b).

The Hall effect, discovered by Edwin Hall, states that when current flows through a conductor in the presence of a perpendicular magnetic field, the charged particles flowing through it are deflected to the side due to the Lorentz force, taking a curved path and creating a small measurable voltage across the conductor, as shown in Figure 2.7 (Texas Instruments, 2021).

The Hall voltage is directly proportional to the strength of the magnetic field and the current, so a stronger magnetic field will result in a higher voltage. Just like with accelerometers and gyroscopes, each magnetometer can only measure the magnetic field along one axis, so three are usually packaged perpendicularly to measure the three-dimensional magnetic field vector (UAV Navigation, 2025).

Since the Earth's magnetic field points north, the magnetometer can determine its yaw (or heading) relative to true north. Pairing a magnetometer with an accelerometer will result in being able to obtain information on all three rotational axes.

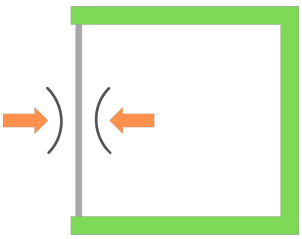


Figure 2.8: Schematic of a MEMS barometer, showing the sealed cavity and the flexible membrane (Own Illustration).

A great problem with Hall-effect magnetometers is their sensitivity. As the Earth's magnetic field is relatively weak, ferromagnetic objects near the sensor easily distort its readings by altering the magnetic field around them, rendering it useless. Another issue with this type of sensor is that, since Hall voltages are typically in the µV (10^{-6}) range, whereas USB ports operate at $5\mathrm{V}$, for example. Since the magnitude of the measured voltage is so low, external factors such as thermal noise or geometric imperfections on the sensor can very easily distort the magnetometer's reading (VectorNav, 2025b).

2.2.4 Barometric Altimeters

A barometric altimeter is a sensor that measures altitude through air pressure. It operates on the principle that atmospheric pressure decreases with altitude, as there is less air above the sensor weighing down on it. Near sea level, this drop is approximately $12\,\mathrm{Pa}$ for each meter of altitude gain. Formally, this change of pressure is given by the following formula:

$$P(h) = P_0 \cdot e^{\frac{-mgh}{kT}}$$

where:

- $P(h)$ is the pressure at a given altitude h,
- P_0 is the reference pressure at sea level, approximately $101\,\mathrm{kPa}$,
- m is the mass of an air molecule,
- g is the acceleration due to gravity,
- k is Boltzmann's constant,
- T is the absolute temperature.

While the relationship between altitude and pressure is exponential, it can be linearly approximated for short ranges. Thus, most barometric altimeters rely on lookup tables for certain pressure values, and interpolate height estimations between them.

Once again, most barometric altimeters employ MEMS technology due to its low cost and small size. These sensors typically contain a sealed cavity at a specific pressure and a flexible diaphragm that deforms in response to changes in air pressure. This deformation can then be detected through capacitive or piezoresistive sensing, allowing the device to measure pressure with high precision (Avnet Abacus, 2025).

One major drawback of barometric sensors is that their readings are affected by environmental pressure changes. For example, storms or passing weather can alter the atmospheric pressure, which can result in different altitude readings from the same location at different times. Without correction, these changes in pressure could be misinterpreted as a climb in altitude.

To counteract this variability, most flight controllers operate on relative altitude. Instead of relying on a fixed reference such as sea level, the sensor records the pressure before taking off, and then measures changes relative to this baseline (Betaflight Dev Team, 2025). This is useful in applications such as drones because most use cases only require a relative altitude to be able to avoid the ground and other obstacles. However, while this approach cancels most long-term offsets, rapid fluctuations from airflow or weather can still disturb the readings. Despite these setbacks, modern consumer-grade barometers such as Infineon's DPS368 can achieve resolutions of up to $2\,\mathrm{cm}$ for relative altitude or $8\,\mathrm{m}$ for absolute altitude (Infineon Technologies, 2020).

On drones, one of the biggest issues for barometric altimeters is propwash, which is the disturbed airflow generated by the propellers. This turbulent air can cause rapid changes in pressure around them, which overwhelm the subtle variations caused by altitude changes. To counteract this, barometers are often placed at a distance from propellers, and they are covered by the fuselage or other protective elements such as foam to reduce the impact of the airflow (Betaflight Dev Team, 2025).

2.2.5 Time-of-Flight Altimeters

A time-of-flight (TOF) altimeter determines altitude by measuring the amount of time it takes for energy to travel from the sensor to the ground and back. Its name comes from measuring how long it takes for the energy to "fly" from and to the sensor. Depending on the sensor, this energy can be in the form of ultrasonic waves, infrared, or laser (Toa & Whitehead, 2019).

Since $distance = time \times speed$, an equation can easily be derived to determine altitude:

$$d = c \cdot \frac{t}{2}$$

As the length of time t registered spans both the travel from the sensor and back to it, this equation first divides it by two to obtain the amount of time taken for the energy to reach the ground, and then multiplies it by the speed of light or the speed of sound, depending on the type of sensor.

Ultrasonic TOF sensors are relatively inexpensive and ideal for short ranges, ideally under 10 meters. They are useful for low-altitude stabilization and automated landings. Alternatively, optical sensors such as infrared or laser can achieve longer measuring ranges with higher precision, with the downside of a sharp decline in reliability when conditions are not ideal (Toa & Whitehead, 2019).

TOF sensors are largely immune to weather-induced variations, as they don't rely on atmospheric pressure, making them an ideal candidate for flight. However, they are still prone to interference. Irregular surfaces or rainfall may affect optical sensors, while temperature and propeller noise can disturb ultrasonic sensors.

Despite optical sensors having an augmented range, in consumer drones, they are still mostly limited to about 15-20 meters, which is enough for landing and low-altitude hovering, but not reliable for higher-altitude navigation (DJI, 2025).

In practice, most drones combine both TOF and barometric altimeters. A flight controller may rely on a barometric altimeter while flying high above the ground, then transition to TOF-based measurements for precise landing control. This approach capitalizes on the strengths of each sensor and allows for reliable altitude estimations throughout the flight.

2.2.6 GNSS

The Global Navigation Satellite System (GNSS) is a global positioning system that employs satellites to determine the user's position through trilateration. Various regions and countries operate their own groups of satellites, known as constellations, which are part of the global GNSS network. For example, the European Union runs Galileo with 28 satellites (European Space Agency, 2025), the United States operates the more broadly known GPS with 31 of them (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2025b), Russia maintains GLONASS, and China provides BeiDou (BDS). All of these systems are free for public use, however they also include encrypted signals for restricted military use. For simplicity, this section will cover information specifically extracted from GPS, but these principles apply to all constellations.

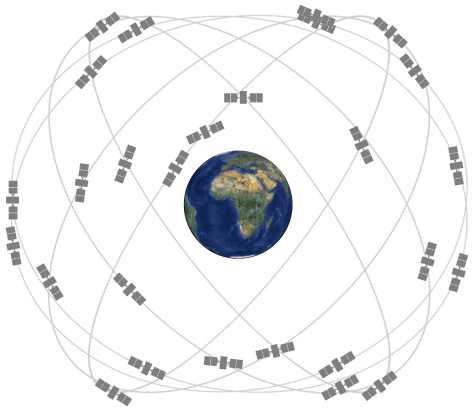


Figure 2.9: Official depiction of the United States' GPS constellation (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2025c).

Before a receiver can calculate its exact location through GNSS, each satellite must first know its own position and orbit with great precision. To do this, a series of monitoring and control stations are scattered around the globe. These stations are in charge of determining each satellite's position and trajectory, and relaying this information to the satellite, along with precise timing measurements to adjust the atomic clock onboard the satellite and thruster directives to stay on trajectory (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2025a).

In space, the satellites follow these determined orbits at around twenty thousand kilometers of altitude in medium Earth orbit (MEO), circling the Earth twice per day. These orbits are designed such that, on any point on the surface of Earth, at least four satellites are visible at any time. To do this, a minimum of 24 satellites is required, though more are often in orbit to account for maintenance and general downtime (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2025c). These orbit trajectories can be seen in Figure 2.9.

Once the satellite has its position information, it periodically broadcasts it along with a precise timestamp. On the ground, receivers such as those found on phones capture this data. By comparing the broadcast's timestamp and the device's timestamp, and since radio waves travel at the speed of light, we can find the distance from the satellite to the device similarly to how TOF altimeters measure it, as explained in the previous section. Since we are in three dimensional space, this creates a sphere around the satellite with its radius being the measured distance, where the device could be anywhere on the surface of this sphere. To reduce the amount of possible positions, data is taken from another satellite, creating a second sphere, and noting where both spheres intersect. This diminishes the amount of possible locations, but there are still multiple, so a third satellite is used to reduce the number of possible locations to two. Since one of these positions is typically out in space, it is discarded. Thus, the remaining

position where all spheres intersect is the device's location. Additionally, a fourth satellite is used to correct timing errors, as most devices' clocks are not as precise as the atomic clocks operated by the satellites [1]. Since this process uses distances to calculate the position rather than angles, it is called trilateration, not triangulation (GIS Geography, 2018).

GNSS is especially useful due to its ability to work all around the globe at any altitude with moderate precision. GPS can typically reach an accuracy of less than $2\,\mathrm{m}$ (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2025d), while Galileo can reach $1\,\mathrm{m}$ of precision (European Space Agency, 2025). Although it is not as precise as,for example, a barometric altimeter, its absoluteness and resistance to drift and meteorological effects make it a useful sensor for flight. GNSS is also highly applicable, as it has global coverage. While it may not be able to individually navigate low flight or precise positioning, it can be used for general guidance through open air, as well as crash recovery. If the quadcopter crashes, GNSS can be utilized to determine its last position and recover it manually.

2.2.7 Sensor Fusion

As discussed in the previous sections, no sensor is without its flaws. While they all provide valuable data, they are not able to compute the position of a drone individually. Gyroscopes measure angular velocity and therefore cannot measure "absolute" orientation, barometric altimeters are easily affected by weather, TOF altimeters cannot operate beyond a small altitude, and GNSS provides a lacking resolution for precise maneuvers.

Sensor fusion is the process of combining readings from multiple sensors to combat each one's weaknesses. For example, the drone might use its accelerometer to determine the direction of gravity, and therefore the ground's, so that the gyroscope can determine its absolute orientation more reliably, or the barometric altimeter may be calibrated through the GNSS receiver to obtain absolute altitude, while still maintaining a high resolution. By blending all these data points, the system creates a better image of the drone's state than any individual sensor could provide (Elmenreich, 2002).

In practice, this fusion is achieved through filtering and estimation algorithms, which determine how much trust or weight to give to each sensor based on the task at hand, such as prioritizing a barometer over GNSS for small movements. For instance, the magnetometer is usually trusted for absolute heading, and is typically used to correct the drift of the gyroscope, which tracks short-term changes. However, if the magnetometer reads a sudden change in heading that

[1] A visual, more detailed explanation of this process, starting with a 2D example and later on applying it to 3D space, can be found here.

the gyroscope does not, the system may treat it as magnetic interference from nearby objects, reducing its weight dynamically. Additionally, state estimators such as the Kalman Filter will implement an expected, theoretical state as well as sensor data, to reduce noise as much as possible.

2.3 CONTROL THEORY

All drones and autonomous systems rely on control theory to function. Without it, the drone would not be able to hover, follow a path, or react to disturbances. Control theory is the framework for understanding and modeling how a system behaves through feedback, formalizing how the system currently is, how it compares against its expected state, and what corrections should be applied to return the system to its expected state. This section covers the fundamentals of control theory, the types of control systems and how these concepts apply to drones.

2.3.1 What is a Control System?

A control system is a set of devices, components and processes that direct the behavior of other systems. Its goal is to make a system behave in a desired manner, despite disturbances or a change in conditions. Control systems are present in most modern technology, from household appliances to industrial machinery, as they are responsible for ensuring systems operate predictably and accurately. Control systems are composed of multiple interconnected components, with specific roles in regulating the system's behavior (Electronics For You, 2023).

The Plant is the physical system being regulated by the control system. It responds to external inputs, and it evolves over time based on several factors and characteristics, which determine the difficulty of controlling it and the most suitable methods for doing so. Consider a room with an Air Conditioning system installed within. The AC unit receives signals based on the built-in thermometer's readings, and regulates the cooling to reach the desired temperature. In this example, the process of regulating the room's temperature as a whole is the control system itself, while the room (or the air within) is the plant, as it is the object of the temperature regulation, with an example of a characteristic that could affect the system over time being the room's size, increasing or diminishing the time it takes for the room to cool down (Simrock, 2011).

In order to regulate the plant, a Control Signal (sometimes referred to as Input) is required. In the previous temperature regulation example, the control signal would be the electrical signal going to the AC instructing it to turn on or off (Simrock, 2011). However, the signal itself does not act on the environment. Instead, it is translated into a physical effect by Actuators, which are any component that has a repercussion on the plant, such as a motor on a drone, or the AC itself.

Conversely, the Output is the resulting effect on the plant due to the actuator. Outputs can be measured through sensors to obtain information on the system's performance. In the previous example, the output would be the room's temperature itself (Simrock, 2011).

A Disturbance is any external factor that unexpectedly affects the plant. Disturbances can be both positive or negative, but they are unpredictable and difficult to account for. For example, sunlight coming through a window could benefit a heating system, but it could also harm an air conditioner's performance. Depending on a system's type (explained in the next section), it may or may not be able to react and compensate for disturbances (Simrock, 2011).

In drones, control systems are necessary for hovering, maintaining an altitude, staying upright or even following paths. Control signals to the motors control thrust and maneuvers, while outputs measured by sensors indicate its position, speed, and altitude. Lastly, control systems are also often utilized to deal with disturbances such as wind or turbulence.

2.3.2 Open and Closed-Loop Systems

Control systems can be generally categorized into open-loop and closed-loop, depending on whether they use feedback to adjust their behavior or not.

Open-Loop systems operate without monitoring the results of their actions. It applies inputs to the plant through a set of instructions, without verifying that the desired output is met. For instance, most traditional washing machines are controlled by an open-loop system, as they spin and add water or detergent purely on a timer. The washing machine assumes that this pre-programmed sequence will clean all clothes regardless of initial conditions such as the amount of dirt on them. If these initial conditions are not as expected, the system will not be able to compensate for the additional amount of dirt and will result in unclean clothes at the end of the washing cycle (Lisleapex Electronics, 2024).

In contrast, Closed-Loop systems (or feedback systems) measure the output and compare it to the desired state. This information is then used to adjust the controller's input, in order to minimize the difference between the current and desired outcome. This feedback is achieved through various sensors measuring outputs relevant to the system's objective, and it is what allows closed-loop systems to adapt and counteract disturbances. An example of a closed-loop

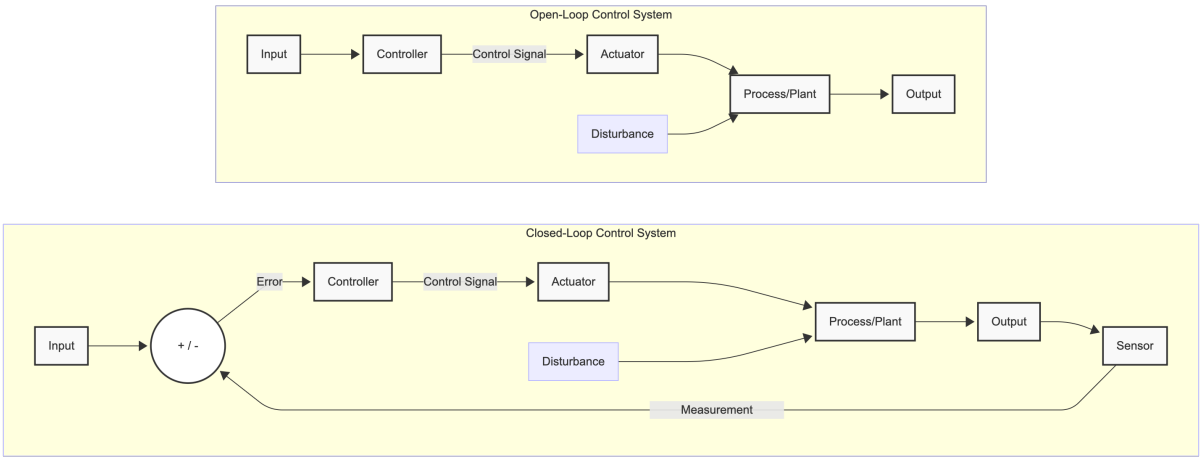


Figure 2.10: Flowcharts demonstrating the operation of open-loop (top) and closed-loop (bottom) control systems (Own Illustration).

system is a car's cruise control. Through a speed sensor, the car compares its real and desired velocity, and adjusts the throttle accordingly to cruise at a constant speed. Therefore, even if the car goes up a hill, the system will adjust its power output to ensure it does not slow down (Lisleapex Electronics, 2024).

Figure 2.10 provides a visual overview of both types of control systems. In this diagram, the "Input" refers to the goal given to the control system by the user, not to be confused with the control signal.

Both of these systems have specific use cases where they are useful. In a washing machine, it is typically not necessary to check how much detergent is in the water and adjust the amount dynamically, as a set quantity will be sufficient for most loads. However, if a car were to go downhill while on cruise control, this may result in it reaching an excessive speed, which may be dangerous to the passengers and other drivers on the road. Open-loop systems are typically implemented in applications where precise control is not necessary or where disturbances are uncommon, while closed-loop systems are useful where precision and adaptability are required (RT Engineering, 2023).

In drones, closed-loop systems are essential for flight. Through the various sensors covered in Section 2.2, the drone continously adjusts orientation, altitude, and velocity in real time. Without the feedback of closed-loop systems, external disturbances such as wind would greatly impact the drone's ability to perform its objective (Zheng et al., 2021).

2.3.3 Setpoint and Error

The Setpoint is the desired value of a system. It represents the target condition that the system is designed to achieve, such as a specific temperature, velocity, or altitude. Formally, it is expressed as $r(t)$, where t is time, because the setpoint can vary dynamically depending on the application.

The Error quantifies the difference between the desired setpoint and the actual output. It is only present in closed-loop systems, as open-loop systems have no way to measure the output, and it is the metric upon which the controller acts and is evaluated on. If said output is represented as $y(t)$, then the error function can be expressed as:

$$e(t) = r(t) - y(t)$$

This mathematical function is useful because it gives a measure of system performance at each instant. If the error is positive, then the output is below the desired value, while a negative error signifies that the output has exceeded the setpoint. Once the error is known, the controller applies a control strategy that determines how the error is analyzed and handled.

In drones, the setpoint and error may refer to its orientation, altitude, speed, or location, among others. Typically, the flight controller will manage various control algorithms to individually deal with each magnitude. The concept of the setpoint as a function is also especially clear in this context, as drones may have a set of waypoints to navigate to. When a waypoint is reached, the setpoint changes to be the location of the next one, so it is dynamically changing.

2.3.4 System Stability

System stability is one of the core concepts of control theory. A system is considered to be stable if its output behaves predictably and in a bounded manner when subjected to a bounded input. Simply put, this means that the output will stay within a specific range so long as the input external to the system does the same.

There are various types of stability, varying in details. Bounded Input-Bounded Output (BIBO) stability requires the system to remain bounded as described in the previous paragraph. Asymptotic stability, apart from implying BIBO, additionally requires that the output converges to a certain point over time. In contrast, while marginally stable systems also imply BIBO, they do not converge nor diverge, such as oscillating within a range (Franklin et al., 2019).

When analyzing a system's stability, there is a clear distinction between short-term behavior and long-term behavior. The Transient response of a system refers to its short-term reaction to a change in input or disturbance. During this period, the controller may overshoot the setpoint or oscillate before settling to it. Additionally, Settling time is used to describe the time required for these deviations to stop and the system to return to a tolerance range around the setpoint (Franklin et al., 2019).

An important metric to analyze the long-term performance of a stable system is its Steady-State Error (SSE). After the transient response period has passed, some systems may still have a slight deviation from their setpoint. Steady-state error is formally defined as the limit of the error function as time approaches infinity, or simply the amount of error present after an extended period of time:

$$e_{ss} = \lim_{t \to \infty} e(t)$$

Where $e(t)$ is the error function described in the previous section. By evaluating the SSE, different control strategies can be analyzed in order to find which one minimizes said metric (Tutorialspoint, 2025).

Settling time and SSE often involve a trade-off. By prioritizing a shorter settling time, the steady-state error will typically rise, and prioritizing a lower SSE may result in sluggishness during the transient response period. In drones, a flight controller that is tuned to react too aggressively may risk instability such as oscillations, while an overly conservative controller may struggle to react to disturbances such as wind (Monolithic Power Systems, 2025).

2.4 CONTROL STRATEGIES

A control strategy is the algorithm or method through which a control system's controller interprets the system's output, compares it to the desired setpoint, and computes the necessary input to guide the plant toward said setpoint. Strategies vary in complexity, from simply turning on or off to highly sophisticated algorithms involving dynamic models or advanced mathematical tools such as those found in calculus.

Control theory is applicable to a wide range of use cases. Various systems may prioritize speed, precision, energy efficiency, or disturbance protection, among others. No single approach is universally valid across all applications, which is why a variety of control strategies are employed in real scenarios.

The effectiveness of each of these strategies can be assessed through metrics such as stability, transient response, settling time, or steady-state error, as discussed previously. The following subsections will detail several common control strategies and their possible use cases.

To define each type of control, $u(t)$ will be used to denote the control signal function, which represents the controller's output applied to the plant. Additionally, $e(t)$ will be used to denote the error function, the difference between the setpoint and the current state, as introduced and defined in Section 2.3.3.

2.4.1 On-Off Control

On-Off (or Bang-Bang) control is the simplest form of control strategy. The controller output can only abruptly switch between two values depending on if the error is positive or negative:

$$u(t) = \begin{cases} u_{\max}, & e(t) > 0 \\ u_{\min}, & e(t) < 0 \end{cases}$$

Recall that $e(t) = r(t) - y(t)$, so if the plant is above the setpoint, the error will be negative, and if it is below the setpoint it will be positive (WPILib, 2025).

In practice, on-off controllers are typically implemented with a deadband to reduce excessive switching. If $|e(t)|$ is within the deadband (i.e. smaller than a chosen tolerance), the controller will not react, setting $u(t)$ to 0. This can differ from $u_{\min}$, which could represent a negative compensation such as reverse thrust on a quadcopter, while the deadband would turn the motors completely off. If $u_{min} = 0$, then the deadband prevents excessive wear on the actuator due to constant on-off cycling (Evans, 2023).

A classic example of on-off control is a thermostat. Thermostats usually turn the heater on when the room is too cold, and fully turn off when the temperature exceeds the setpoint. A deadband is often implemented as well to prevent wear on the system and avoid rapid cycling (Evans, 2023).

On-off control is typically favored in applications with low complexity and cost. It requires no mathematical modeling of the system and is easy to implement software-wise. However, its binary nature makes it imprecise, and it tends to cause oscillations around the setpoint since the controller cannot fine-tune its response (Bold City Heating & Air, 2025).

While this control strategy is useful in heaters or refrigerators, it is not suitable for drones. Flight requires continuous and precise adjustments to thrust and orientation, but on-off control would produce large, detrimental oscillations, leading to instability (Control Station, 2015).

2.4.2 Proportional Control

Proportional control is one of the most common control strategies employed. In this strategy, the controller's output is directly proportional to the system's error:

$$u(t) = K_p\, e(t)$$

This linear relationship between the control signal and the error means that the larger the error is, the larger the output will be (x-engineer.org, 2025).

In the previous equation, K_p is the proportional gain of the system. While it technically doesn't need the "proportional" distinction here, it will be useful for differentiation in the following control strategies. The gain acts like the slope of a linear function: it scales the output by multiplying the error. A low gain produces sluggish responses, while a high gain can cause overshoot or oscillations. In cases where K_p is extremely large, the system behaves similarly to an on-off controller (x-engineer.org, 2025).

Despite its simplicity and ease of implementation, proportional control is not capable of eliminating steady-state error. This is because, as the system approaches its setpoint and the error diminishes, the control output decreases, oftentimes leaving a small deviation from the desired value (Haber, 2024).

In drones, proportional control can be applied for simple corrections such as pitch, yaw or roll, often with a lower gain to reduce overshoot. However, the presence of steady-state error means that more complex strategies, such as PID control, are often preferred (Sayed, 2024).

2.4.3 Proportional-Integral-Derivative Control

To overcome proportional control's limitations, such as steady-state error and excessive oscillations, Integral and Derivative terms are introduced into the equation.

In mathematics, an integral from a to b of a function represents the cumulative area under its curve between those two points. In control theory, taking the integral of the error function means that it is possible to keep track of errors over time:

$$u_i(t) = K_i \int_0^t e(\tau)\, d\tau$$

This term tracks the cumulative error of the system from its start to the current time and acts against it.Therefore, if a small error persists for a long time, the integral term will keep growing until it forces the system to correct it, effectively canceling out steady-state error. Additionally, the integral gain modulates how aggressively this term affects the output: a low K_i results in slow compensation, while a higher K_i can result in oscillations once again (Duckietown, 2025).

Conversely, the derivative is a tool to calculate the rate of change of a function. If $e(t)$ is increasing, this must mean that the plant is falling below its setpoint. Since the derivative of an increasing function is positive, the derivative term will add to the controller's output, bringing the plant closer to its setpoint, and the opposite process will happen if $e(t)$ becomes negative, lowering the output:

$$u_d(t) = K_d \frac{d}{dt} e(t)$$

By "predicting" where the system is going, this term dampens oscillations, reduces overshoot and overall stabilizes the system during rapid changes. If the gain is tuned too high, the term may respond too aggressively to sensor noise, and, if not big enough, it may not be able to respond to oscillations properly (Duckietown, 2025).

PID control combines the Proportional, Integral and Derivative terms into the output function by adding them together:

$$u(t) = K_p\, e(t) \;+\; K_i \int_0^t e(\tau)\, d\tau \;+\; K_d \frac{d}{dt} e(t)$$

In this strategy, the proportional term gives immediate response to the current error, the integral term responds to the cumulative error to avoid steady-state error and the derivative term anticipates future error to prevent overshoot. Combined together, they allow for precise control suitable for a wide range of scenarios.

Finding appropriate values for K_p, K_i and K_d is often done through one of two methods. The "guess-and-check" method involves selecting arbitrary values for these parameters and slowly adjusting them based on system response. The Ziegler-Nichols tuning method involves setting the integral and derivative gains to zero, and finding a value for the proportional gain such that it oscillates consistently. Then, this proportional gain is ran through tabulated equations, computed by John Ziegler and Nathaniel Nichols, giving an appropriate value for K_i and

K_d. These parameters should result in a 75% reduction of the amplitude of each oscillation compared to the previous one, although many systems often benefit from additional tuning past these values, as this is considered an aggressive tuning method (National Instruments, 2020).

PID control is widely favored due to its versatility, as it is able to maintain stability in a variety of systems. However, it is more complex to implement than other strategies, and careful tuning is required for optimal performance.

In drones, PID controllers are the most used strategy. They are often used to regulate orientation as well as altitude, and they are especially useful in holding a position in space due to their reduction of steady-state error. Since integrals and derivatives require complex systems to properly compute and drones have limited computational power, these functions are often approximated (Bayisa & Geng, 2019).

3 PRACTICAL FRAMEWORK

You cannot learn to swim by reading a book. To do so, you have to get into the water.

Shiv Khera

The purpose of this practical framework is to apply the theoretical concepts covered in the previous section, and observe how they interact with real, physical systems. These concepts are explored through two separate simplified experimental setups to replicate common drone dynamics scenarios in order to show how they are handled in real unmanned autonomous aerial vehicles. Additionally, a Python simulation of drone dynamics is included after these physical experiments, to discuss theoretical concepts.

The first experiment is a rotational oscillator, designed to model the rotational dynamics of a drone around one axis. The second experiment is a vertical oscillator. This model is fixed in orientation and has the objective to hover at a specified setpoint, having to moderate its thrust in order to do so. These rigs employ the sensors covered in the Section 2.2, and they serve to demonstrate how parts interact with each other and how to pick components for maximum ease of operation.

This section is first divided into two main parts corresponding to each rig. Each part details the objectives of its specific experiment, the mechanical and electronic design, the sensors and components used, and the experimental results obtained. Lastly, the Python simulation covers phenomena present in control systems, as well as showing the difference between the real state of a system and the system reported to the controller by the sensor.

3.1 ROTATIONAL OSCILLATOR

This first experiment seeks to understand how autonomous quadcopters handle stabilization in their rotational axis, namely their pitch and roll. By freely allowing rotation in one axis but restricting the others, the model is used to experiment with the sensors pertaining to angular orientation detection and the various applicable control strategies, analyzed through metrics such as steady-state error, covered before.

3.1.1 Mechanical and Electronic Design

In order to achieve the desired rotational motion, two brushless motors equipped with 5" propellers are mounted at opposite ends of an aluminum rectangular hollow section tube, acting as the main support arm. The tube measures $500\,\mathrm{mm}$ in length with a $20 \times 10\,\mathrm{mm}$ cross-section, in order to reduce weight but maintain enough surface area to comfortably mount components. This symmetrical configuration ensures balanced thrust distribution for both directions of rotation.

At its midpoint along its length, a circular aluminum tube is attached perpendicularly, serving as the rotational shaft. The shaft has an outer diameter of $16\,\mathrm{mm}$, and passes through two KFL000 flange bearings positioned on opposite ends of the supporting frame, allowing free, low-friction rotation along one axis while constraining the others. Aluminum was chosen for both tubes due to its low weight and high stiffness, in order to provide rigidity while reducing inertia.

For structural stability, the bearing housings are mounted on a U-shaped wooden frame, constructed from three perpendicular planks joined with screws at their intersections. The frame provides a rigid base to fix the experimental rig.

To connect both aluminum tubes and house the electronic components, a custom 3D-printed mounting bracket was designed using Tinkercad and printed on an Ender 3 Pro. The bracket was printed in PLA at standard resolution. This material was selected due to its accessibility and ease of printing. Both aluminum tubes are press-fitted directly through the printed structure in their respective orientations, to ensure an accurate alignment between the support arm and the shaft. This required setting specific tolerances on the bracket, to ensure enough friction existed to prevent the tubes from accidentally moving.

The bracket provides a compartment for the battery as well as an open design to more easily service or replace components and disassemble as needed. It also provides sufficient clearance to ensure all components are protected from the propellers in the event of an accident. Due to its flat, extensive surfaces, sensors can be mounted in any desired orientation, which can be crucial for certain ones. This bracket can be seen in Figure 3.1, which shows the fully assembled rig. Another benefit of this bracket is the total shielding of the battery from the propellers. Lithium-polymer batteries (such as the one used) are prone to fire with minor damage, so it is essential to shield them prom possible impact.

The electronic system of the rotational oscillator provides the necessary actuation, sensing and control logic functions that allow the rig to replicate drone dynamics. This system is designed around an ESP32 microcontroller, chosen for its built-in Wi-Fi capabilities and extensive PWM outputs. Additionally, this processor retains compatibility with Arduino libraries while being up to 15 times faster (Sodhi et al., 2024), allowing for more computationally-demanding tasks.

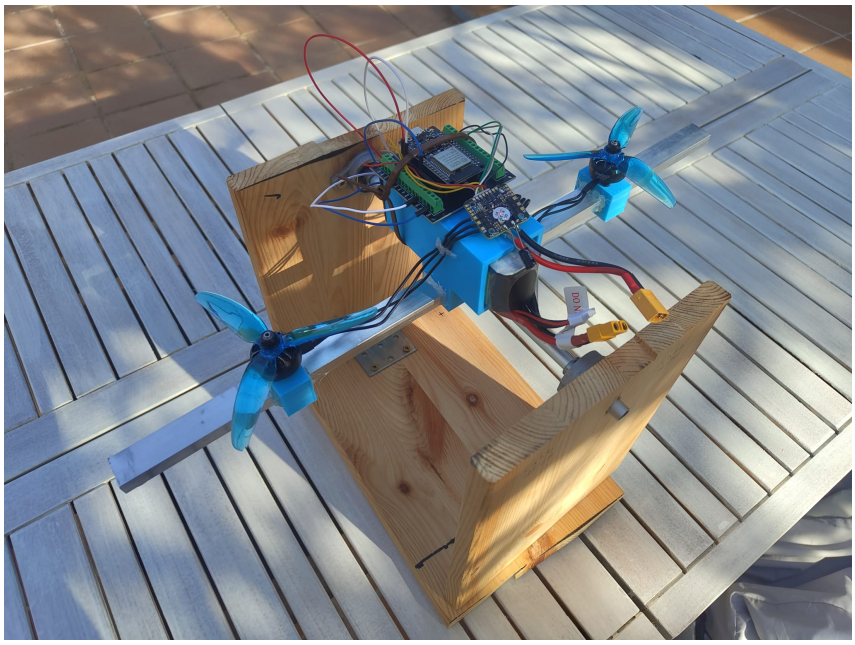


Figure 3.1: Fully assembled rotational oscillator rig. More images of the rig are available in the appendix (Own Photograph).

Actuation is carried out by two `iFlight XING-E PRO 2207 1800KV` brushless motors, commonly found in quadcopters, placed on opposite sides of the rectangular aluminum tube. They are driven by a $40\,\mathrm{A}$ 2-in-1 Electronic Speed Controller (ESC) unit, which integrates two motor drivers into one module to simplify wiring and reduce power differences between both motors. The ESC receives control signals directly from the ESP32 through Pulse Width Modulation (PWM) signals. To attach the motors to the rectangular bar, a second custom 3D-printed bracket was designed. This bracket contains holes for the motor's screws with stoppers for their heads, as well as holes on the opposite side to be able to screw and unscrew them with a hex key. Perpendicular to the mounting holes, space for the rectangular bar was created, so that it could be press-fitted through, having the screw heads be sunk into the piece enough so as to not obstruct the aluminum bar.

Power is supplied by a 6S (six cells in series) lithium polymer battery rated at $1200\,\mathrm{mA\,h}$ with a discharge rating of 150C [1], meaning that the battery could theoretically supply up to $180\,\mathrm{A}$, exceeding the maximum draw of the chosen ESC, reducing the risk of a lithium polymer fire. While the battery is directly connected to the ESC, the ESP32 draws its power from one of the ESC's integrated $5\,\mathrm{V}$ BECs, for stable and regulated power even at maximum current draw (JHEMCU, 2025), to ensure the microcontroller doesn't freeze and lose control while the motors are spinning. Any sensors that may be used are then directly powered by the ESP32's high voltage pins.

[1] Note that "C" does not refer to the SI unit coulomb, rather, it is a dimensionless rating that indicates how quickly a battery can be safely discharged relative to its capacity.

Rotational motion is measured with a SparkFun BNO086 Inertial Measurement Unit, which hosts a magnetometer, an accelerometer and a gyroscope, as well as a processor. For this experiment, only the gyroscope and accelerometer components will be used to monitor the rig's angular orientation. Due to the processor mounted on the chip, angular orientation can be easily determined. Rather than having to manually implement the computation for the orientation of the sensor through methods such as those explained in Section 2.2.1, the processor handles this task, allowing for a simple call for its orientation when needed instead (SparkFun Electronics, 2025). The IMU is mounted onto a foam block to reduce possible vibrations.

To prevent accidental toppling, the rig was operated under dual operator supervision, with one person monitoring the interface to digitally stop the experiment in the event of an emergency and another to physically stabilize the rig and manually restrain it.

All control logic is implemented and run on the ESP32 itself, without the need for external computation. The board hosts a minimal web interface accessible over Wi-Fi, which allows the user to start or stop the experiment (avoiding having to dangerously disconnect the battery while the experiment is ongoing), tweak parameters as needed, and download the gathered data for further analysis. During operation, the ESP32 continuously records sensor data and control signals into its internal memory, which can then be exported onto the user's device.

As mentioned before, the electronics are compactly housed within the 3D-printed mounting structure. Specifically, the ESP32 and ESC are both attached to the top, while the sensor lays at its back and the battery is securely fastened within the mount. This design allows for flashing new code to the ESP32 without having to remove all components.

3.1.2 Firmware and Software Implementation

The firmware to control the rotational oscillator is implemented as a PlatformIO project for the ESP32, written primarily in a `main.cpp` file that contains the control logic, IMU interfacing, PWM generation, experiment timing, and data logging. A secondary file, `web.cpp`, implements the web interface. It is important to note that this web functionality was generated with the assistance of an AI tool (Google, 2025). However, this is not the case for the rest of the code, such as the control strategies and the interfacing with the sensor and actuator. This is because the web functionality falls outside the scope of the project, namely control theory and its implementation. The code relies on the `SparkFun BNO08x Arduino Library` for interfacing with the BNO086 IMU over I^2C, and on the ESP32 `WebServer` library for the hosted

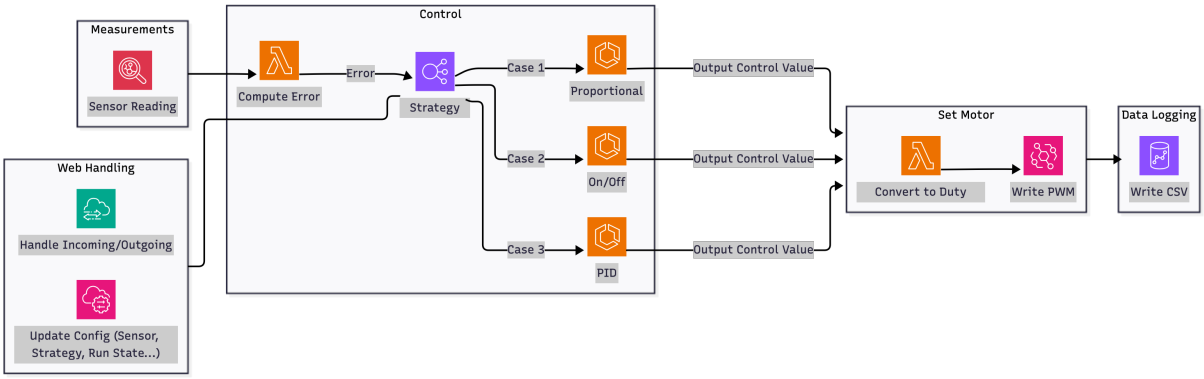


Figure 3.2: Block diagram of the control loop architecture, showing the flow from web handling and sensor measurement to control strategy selection, motor actuation and data logging. Each node indicates a processing stage. For every iteration of the loop ran by the ESP32, the entire sequence of operations shown in the flowchart is performed (Own Illustration).

user interface. The system has the capability to operate on four control modes selectable from the webpage: Off, On–Off, Proportional, and PID. The objective of the controller is to use the absolute fused orientation provided by the BNO086 as the process variable and regulate it such that it is in an upright position.

All control executes inside a single main loop. Timing is handled using `micros()` for precision. The effective update rate depends on both the control loop and the BNO086 sensor output rate. Each iteration of the loop performs a servicing of web UI requests, a reading of the fused angle from the IMU, a computation of the error, the execution of the selected control strategy, the conversion of the resulting scalar control output into PWM commands for the two motors, and logging of a CSV containing timestamp, angle, error, selected sensor, control strategy and control output. 1500 rows are stored on the ESP32 at a time and then streamed to the controlling browser and stored in its RAM until they are downloaded from the webpage. An overview of the described loop can be found in Figure 3.2.

The sensor's library provides a high-level call to `getPitch()` to obtain the absolute orientation in the correct axis (note that `getPitch()` is used due to the placement of the sensor. If it were to be placed another way, a call to `getRoll()` or `getYaw()` must be used instead). To decide the setpoint, a calibration routine is triggered through the web interface. During calibration, the system can record either a high or low reading depending on the selected option, and the setpoint is defined as their arithmetic mean:

```
void calibrateHighStep(int selectedSensor) {
    float sum = 0.0;
    for(int i = 0; i < 200; i++) { sum += sensorRead(selectedSensor);
    ↪  delay(10); }
    calibHighVal = sum / 200.0;
    calibMiddle = (calibLowVal + calibHighVal) / 2.0;
```

The `calibrateLowStep()` function behaves in the same way, but saving its measurements to its respective `calibLowVal` variable. The function records 200 measurements and takes their average to ensure sensor noise doesn't interfere in the calibration. Note that `calibMiddle` is a global variable, so that other functions can use it. This is why it is not preceded by a type keyword.

The control architecture consists of several selectable strategies within a `runControl()` function, which begins by calculating the error and then applying the chosen control strategy. Note that the output is later bounded to $[-1, 1]$ by the `setMotorPower()` function even if the control function exceeds these values. In `Off` mode, both motors are kept at the minimum ESC throttle no matter the control output.

In `On-Off` mode, the control output should either be the maximum or minimum, to swing one way or the other, or zero if it is perfectly balanced:

```
if (error > 0) return 1.0;
else if (error < 0) return -1.0;
else return 0.0;
```

As discussed in Section 2.4.2, the proportional control strategy is based on multiplying the error by a constant (its "gain"):

```
return gain_p * error;
```

For the integral control, true mathematical integration is not possible on a microcontroller because it cannot symbolically compute the area under a curve or know the system's exact characteristic equation. Instead, the integral term is approximated numerically using a process known as a Riemann sum (named after German mathematician Bernhard Riemann), which effectively equates to adding up the areas of small rectangles of width dt and height $e(t)$. In this implementation, a right Riemann sum is used. This means that, at each timestep, the error at the current time step is multiplied by the timestep and added to the cumulative total. An example of a right Riemann sum can be seen in Figure 3.3.

Over time, this approximates the area under the error curve, which is the integral of the error. As the timestep gets smaller, the accuracy of the approximation increases. In code, this is implemented as:

```
integralSum += error * dt;
```

In each iteration of the control loop, the current error is added to the sum of the previous errors.

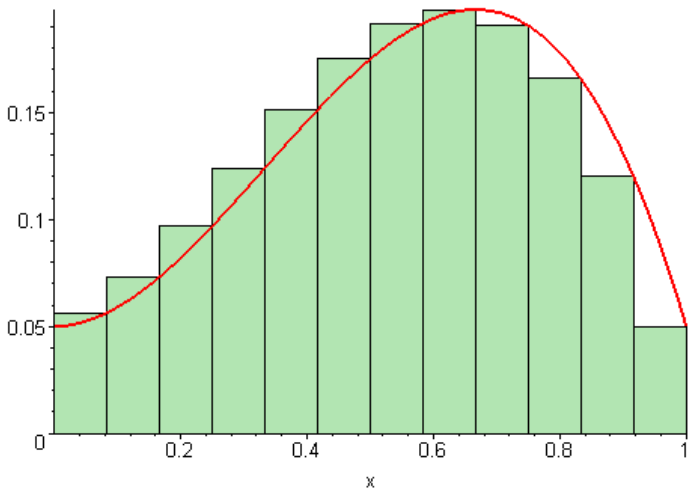


Figure 3.3: An example of a right Riemann sum. Note that this style of sum overestimates increasing functions and underestimates those which are decreasing (09glasgow09, 2009).

For the derivative control, a microcontroller also cannot compute a true derivative in the traditional sense. Instead, a finite difference approximation is used. The derivative term is estimated by the change in error between the current timestep and the previous timestep, divided by the timestep:

$$\frac{d}{dt}e(t) \approx \frac{\Delta e(t)}{\Delta t}$$

This method gives an estimate of how quickly the error is changing at that moment. Like with the integral approximation, the smaller the timestep is, the closer this approximation gets to a true derivative. Again, this is easily implemented in code as:

```
float derivative = (error - lastError) / dt;
```

Finally, for the output, the proportional, integral and derivative terms are all added together:

```
return gain_p * error + gain_i * integralSum + gain_d * derivative;
```

Motor actuation is performed through an ESC driven by PWM at $50\,\mathrm{Hz}$ on ESP32 pins 12 and 13. On startup, the system performs an arming cycle in which both channels are held at minimum duty values for two seconds. A helper function converts the control value into microseconds and a second function updates each PWM channel.

As the experiment runs, each control iteration adds a formatted CSV line into an in-memory buffer. Approximately every $1.5\,\mathrm{s}$ the collected data is streamed to the browser running the UI to avoid overfilling the ESP32's memory. Starting a new experiment clears all previously collected rows. The values are stored as a struct within the code for ease of use and access:

```
struct DataPoint {
    uint32_t timestamp;
    float sensorValue;
    int selectedSensor;
    float error;
    float controlOutput;
    int selectedStrategy;
};
```

The code in its entirety can be found in the appendix or in its GitHub repository.

3.1.3 Experimental Results

The rotational oscillator was successfully assembled and programmed, and initial tests without motors confirmed that the sensing and control functionalities operated correctly. While the motors remained off, the BNO086 IMU provided stable fused orientation readings with expected readings at different angles, with no noticeable drift and minimal noise. Calibrations yielded repeatable setpoints, and the web interface correctly responded to user input. Additionally, motors were able to be controlled through manual power settings, and the system was able to operate solely on the battery it was designed for without the need for external power sources. However, once the motors were activated, the experiment immediately showed severe and unexpected issues that prevented the correct functioning of the system.

As soon as the motors were powered at a sufficient throttle to move the rig, the IMU's output became erratic. Orientation readings fluctuated to values outside of the calibrated range with no relationship to the physical state of the rig. In most trials, the sensor froze entirely and did not return any values. Notably, the problem was present even despite a lack of propellers on the motors, ruling out vibrations or prop wash as the cause for these issues. Even when the motors were turned off again, the BNO086 did not recover from the loss of the communication link.

After repeated attempts to locate the source of the issue, it became evident that this failure was due to electromagnetic interference generated by the brushless motors and the ESC. The high switching currents (up to $40\,\mathrm{A}$) and fast commutation produced electromagnetic fields strong enough to affect the IMU and its communication with the ESP32. The BNO086 communicates through I^2C, a protocol that is relatively sensitive to electrical noise, especially over wires rather than traces on a PCB. Various attempts were made to mitigate this, including adjusting the speed of the protocol, rerouting cables, and even testing alternative communication methods such as SPI and UART. None of these measures showed any improvement, as the sensor failed

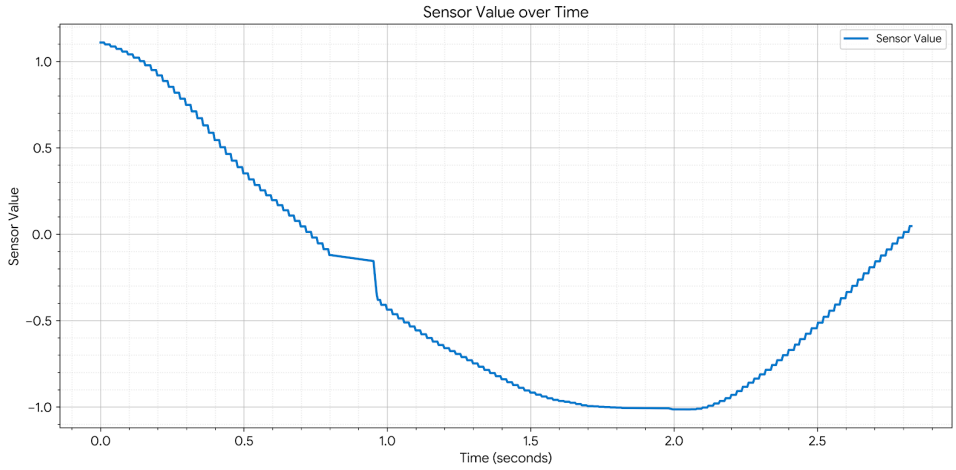


Figure 3.4: BNO086 IMU readings of the rotation of the rig in radians, from one extreme of the rig, to the other, to the center. Note that this rotation was achieved by manually moving the rig, as motor activation effectively disabled the BNO086's communication (Own Illustration).

to respond in all tests after the motors were powered up. Because of the mechanical design of the rig, increasing the physical separation between the motors and the IMU was not feasible without a complete redesign, and even this measure may have fallen short unless a custom PCB were to be designed to avoid cables entirely.

As a result of this interference, no meaningful orientation data could be obtained once the motors were active. Since the purpose of the experiment was to evaluate control strategies and their operation in real-time systems, the lack of measurements made closed-loop control impossible. Since the controller did not receive updated readings, it reacted unpredictably, failing to stabilize in all cases. The consequence of this lack of operability was that no characteristics such as steady-state error, transient response, overshoot, or oscillatory behavior could be observed or recorded. Although the experiment was mechanically and electronically complete, it could not perform its intended function because there was no feedback to base the control loop on while the experiment was running.

This electromagnetic interference occurs due to Faraday's law of induction, which states that a varying magnetic flux passing through a loop induces an electromotive force (voltage):

$$\mathcal{E} = -\frac{d\Phi_B}{dt}$$

In this setup, the high-frequency switching currents (up to $40\,\mathrm{A}$ each) in the brushless motor cables generate a rapidly oscillating magnetic field. The physical separation between the I2C data (SDA and SCL) lines and the ground line creates a conductive loop enclosing an area, which is necessary for magnetic flux to appear. As the motor's magnetic field passes through

this loop, it induces significant voltage spikes on the data lines. Because the change in current is nearly instant, the derivative of the magnetic flux is large, causing a spike in voltage. Due to this speed of the current change, and because I^2C uses a weak 'open-drain' system, these induced voltage spikes easily overpower the signals, resulting in total data loss.

Despite the unfortunate lack of results, the process of developing and testing the rig and its firmware provided me significant insights into the design and creation of control systems. This failed attempt at creating a control system highlighted the importance of sensors in a closed-loop system, and the practical limitations of certain communication protocols in noisy environments.

In summary, the rotational oscillator experiment did not produce meaningful data due to electromagnetic interference from the motors corrupting communication between the sensor and the microcontroller. However, it was still valuable as a learning experience as the creation of this system required matching appropriate components and programming them to interact with each other, and future experiments will be handled with more caution towards electromagnetic interference.

3.2 VERTICAL OSCILLATOR

This second experiment aims to target linear motion rather than rotational motion, specifically vertical movement. In this experiment, the rig is restricted to one degree of freedom along the vertical axis. Therefore, this model utilizes sensors suited to tasks such as hovering and altitude-holding mechanisms.

3.2.1 Mechanical and Electronic Design

To achieve vertical movement exclusively on the vertical axis, the rig employs two $1\,\mathrm{m}$ CK45-C45E chromed steel guide rods with a diameter of $16\,\mathrm{mm}$ mounted on a wooden base (Suministros Miguel Lopez, 2025). These rods support a sliding carriage that moves freely up and down while staying aligned. The sliding carriage consists of a horizontal aluminum beam onto which two `LM16UU` linear bearings are mounted at each end. These bearings provide low-friction alignment along the chromed rods for smooth vertical translation.

To connect the linear bearings to the aluminum beam, a mounting system was adapted from an existing design (Thingiverse Contributor, 2025) and 3D printed with PLA to fit the dimensions and specific mounting necessities of the rig. Additionally, another mounting bracket similar to that of the rotational oscillator was designed to house the electronics.

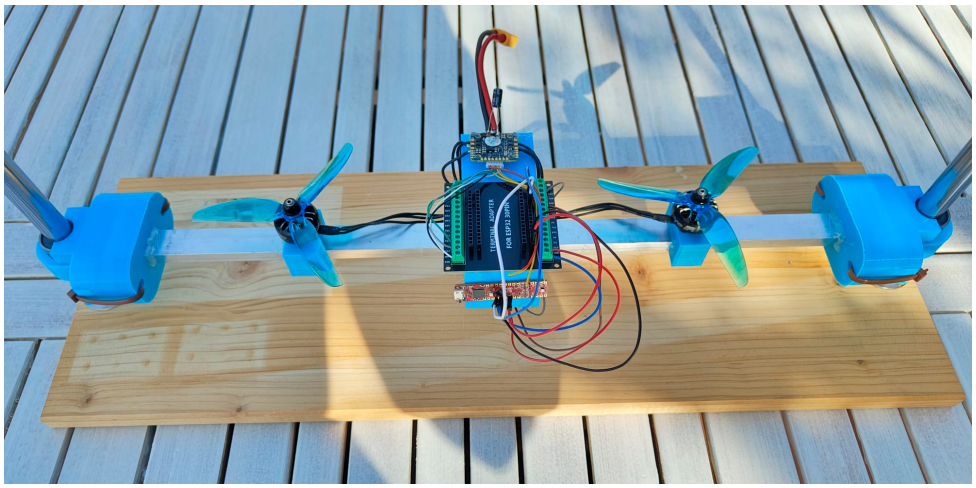


Figure 3.5: Assembled vertical oscillator rig. Once again, more images of this rig are available in the appendix (Own Photograph).

Actuation is provided by two identical motors to those used in the rotational oscillator, mounted upwards to be activated at the same time and create vertical displacement. These motors are also held in place by the same custom mounting brackets designed previously to slide onto the aluminum beam while fixing the motors with the appropriate screws. The battery and ESC are also the same models used on the previous experiment. To prevent the rig from sliding out of the rails, two mechanical stoppers are positioned at the top of the guide rods. The mounting of these components can be seen in Figure 3.5.

Altitude measurement is handled by two sensors mounted onto the custom bracket. A DPS368 barometric altimeter is positioned on the rear side of the bracket, to measure altitude changes based on relative pressure differences. Additionally, a TOF200C[2] time-of-flight distance sensor is attached to the bottom of the bracket, pointing towards the base. This dual-sensor configuration is meant to show the differences in accuracy in both sensors, to understand their uses in commercial drones. A barometric altimeter is expected to clearly under-perform over a range of one meter, but this will serve as an example for the combination of both sensors. However, as the rig will only be a maximum of one meter above the base, the TOF sensor's weakness (namely not being able to reflect its pulse of light on a surface such as the ground) will not be able to be showcased.

An ESP32 microcontroller is mounted at the top of the bracket, similar to how it is mounted on the rotational oscillator. Wires are soldered to the sensors and connected to the microcontroller, and thanks to the I^2C protocol's ability to host multiple devices on the same pins as long as they have different addresses, they do not have to be disconnected to swap between them.

[2]This sensor is also sometimes labeled as a VL53L0X sensor. Despite the difference in nomenclature, they are still the same, and many vendors even print both names on the sensor itself.

3.2.2 Firmware and Software Implementation

The firmware of the vertical oscillator follows the same overall architecture as the rotational oscillator, with most of the codebase being directly reused. The project is once again divided into `main.cpp` and `web.cpp`, with practically the same logic within each one.

The main difference between both rigs is in the form of sensors. While the rotational rig used an IMU, this rig employs both a barometric altimeter (DPS368) and a time-of-flight sensor (TOF200C). The active sensor can be switched dynamically from the web interface, for ease of comparison between both sensors. The barometric altimeter is interfaced through the `Arduino Xensiv DPS3xx` library by Infineon (Infineon Technologies, 2025), the creators of the sensor, while the TOF sensor relies on the `Adafruit VL53L0X` library (Adafruit Industries, 2025), compatible with both branded Adafruit sensors and clone sensors. Both of these libraries use the I^2C protocol to communicate with their respective sensor. As in the rotational oscillator, the system can perform a simple two-point calibration when requested by taking samples and then computing their arithmetic mean for the target height.

Again, the same motor models are also actuated through PWM. However, in this experiment, both motors are powered at the same throttle to achieve translational motion, as opposed to the previous rig, where differential thrust created rotational motion.

Overall, the firmware is nearly identical to that of the rotational oscillator, differing mainly in the integration of new sensors and the fact that symmetrical thrust is required in this rig.

3.2.3 Experimental Results

The initial calibration tests of the vertical oscillator showed that both altitude sensors operated correctly when the motors were powered off. The TOF sensor, in particular, showed consistent measurements to the wooden base, as its 2-meter maximum distance rating easily exceeded the travel of the rig. The barometric sensor also responded appropriately with pressure variations, albeit with lower accuracy. Under static conditions, neither sensor showed much variation, and both were able to complete calibration procedures.

However, just like with the rotational oscillator, as soon as the motors were activated past a certain power threshold, the system encountered the same limitations of the previous rig: the sensors stopped responding to calls for measurements. Regardless of the selected sensor, no data could be gathered once the brushless motors started producing meaningful thrust. Communication was also not able to be restored after shutting the motors off.

Although automatic control was impossible, manual tests showed that the mechanical aspects of the rig functioned as intended. The carriage was able to smoothly glide upwards along the guide rods with minimal friction and no sticking as well as managing to maintain its alignment. Despite this, no evaluation of the control strategies was able to be performed, much like the previous rig, as no feedback signified a lack of error calculation and corrections for On-Off, Proportional, and PID control. Consequently, no behavior such as steady-state error, transient response or oscillation were able to be recorded.

The most plausible explanation for this behavior is electromagnetic interference once again, as the wires for the motors were still in close proximity to the sensors, and wires were used for connecting the sensors to the microcontroller rather than a printed circuit board. As both sensors also employed I^2C communication, a similar electromagnetic force to that of the rotational oscillator was likely created, breaking the communication link to the sensor (For more detail on this force, Section 3.1.3 explains it in detail, covering its physical principles and mathematical representation).

In summary, although the mechanical assembly performed reliably and sensors operated correctly without motor involvement, the experiment could not return usable altitude measurements while actively trying to stabilize. Consequently, no experimental data could be obtained. Nevertheless, the development of this rig (along with the rotational oscillator) did provide valuable insight into aspects such as actuator selection or microcontroller choice. Additionally, extensive research was needed to find a barometric altimeter with sufficient precision (Infineon claim a resolution of $2\,\mathrm{cm}$ for the DPS368 (Infineon Technologies, 2020)). The design and printing of custom PLA parts was also a valuable learning opportunity, especially considering the multiple redesigns for each piece to account for tolerances or to simplify the removal of supports, among others.

3.3 PYTHON SIMULATION

All models are wrong, but some are useful.

George E. P. Box

The two experimental rigs developed in this project, the rotational oscillator and the vertical oscillator, were designed to give an insight into the physical design and behavior of real-time closed-loop control systems. As discussed in the previous sections, both rigs successfully demonstrated the aspects of sensor, actuator, and controller selection, mechanical construction, component integration and embedded development. However, due to electromagnetic interference generated by the motors, neither experiment provided measurements during operation to analyze, so the behaviors covered in the theoretical section could not be evaluated.

To complement the work done on the physical rigs and to be able to show these phenomena, a numerical simulation was developed in Python. The purpose of this simulation is not necessarily to replace the experimental results or to replicate the physical systems with perfect accuracy, but rather to create an environment in which these behaviors can be clearly observed without the unfortunate limitations of real experiments. While the previous experiments mostly focused on practical implementation, the simulation focuses on the response of a control system under idealized conditions. Another benefit of a simulation is that the real state of the system can be known. In a physical rig, the only available data to analyze was the sensor readings and their respective control outputs. However, if these readings were incorrect and the sensor had drift or bias, the control loop could seem to be operating perfectly with no relation to its real-life observed state. In a simulation, even if noise is simulated within the sensor to add realism, the true state can still be obtained, so simulations give more insights into the operation of a control system.

The simulated system represents a simplified rotational oscillator, similar in concept to the physical rig in Section 3.1. A vertical oscillator model is not included, as the dynamics of both systems are fundamentally the same. In the simulation, altitude or angular orientation would be treated as abstract variables that evolve depending on the control loop. Since the sensors are simulated, there would be no difference between barometric altimeters, time-of flight altimeters, or inertial measurement units.

By abstracting away the mechanical implications, sensor limitations and electromagnetic interference from the real systems, the Python simulation allows for the examination of the responses of a system such as those proposed in the physical rigs. This includes the appearance of steady-state error, oscillations, or transient behavior, among others.

The proposed simulation models a second-order system, meaning its dynamics are described by the relationship between position, velocity, and acceleration. Mathematically, the system solves for motion based on the second derivative of position (acceleration). This means that, although it is useful to understand and visualize the described phenomena, it is not a perfect replica of a real system but rather an approximation, especially as it neglects prop wash and other unwanted disturbances in the system.

3.3.1 Simulation Design and Implementation

The simulated system is meant to represent a one-dimensional rotational oscillator, similar to the physical rig built before. In this model, the control signal scaled by a parameter directly acts as the force on the model. This simplified representation allows for clear behavior without the complexity of additional mechanical parameters. A small damping term dependent on velocity is also incorporated to mimic friction, but no higher-order effects are modeled.

To evolve the system over time, the simulation employs Euler's method, a technique for the numerical solution of ordinary differential equations (ODEs). As described by Chapra and Canale (2021), Euler's method is a first-order approximation that predicts a future value by projecting the slope of the function from a known starting point.

This method is derived from the Taylor series expansion of a function $y(x)$ around a point x_i. The exact solution at the next step x_{i+1} is given by the infinite series:

$$y_{i+1} = y_i + f(x_i, y_i)h + \frac{f'(x_i, y_i)}{2!}h^2 + \frac{f''(x_i, y_i)}{3!}h^3 + ...$$

Where:

- y_i is the current value,
- $f(x_i, y_i)$ is the first derivative (slope) of $y(x)$ at x_i,
- h is the step size $(x_{i+1} - x_i)$.

Euler's method approximates this solution by truncating the series after the first derivative term, discarding all higher-order terms:

$$y_{i+1} \approx y_i + f(x_i, y_i)h$$

The difference between the entire Taylor series and this approximation is the Local Truncation Error. While higher-order methods (such as Runge-Kutta) reduce this error by retaining more terms, Euler's method was selected because its computational cost is low and the error is practically negligible for smaller timesteps such as $0.001\,\mathrm{s}$, as every next term will be scaled by 0.001^n.

In this simulation, Euler's method is applied to solve the equations of motion for a rotational system. Time t is treated as the independent variable (akin to x in the previous definition) and the timestep Δt as the step size h.

The update process occurs in two stages for each timestep. First, the angular velocity is approximated at the next timestep, ω_{i+1}, using the current angular acceleration α_i (where $\alpha = \omega'$). Using the previous Euler expression:

$$\omega_{i+1} = \omega_i + \alpha_i \cdot \Delta t$$

Next, the angular position θ_{i+1} is updated using the newly computed angular velocity, ω_{i+1}:

$$\theta_{i+1} = \theta_i + \omega_{i+1} \cdot \Delta t$$

This specific implementation, where the updated velocity is used to calculate the new position, is formally known as the Semi-Implicit Euler method (or Symplectic Euler). In code, for example in a `RotationalOscillator` object, this can be written as:

```
self.angular_velocity += acceleration * dt
self.angle += self.angular_velocity * dt
```

The oscillator is started with an initial angle of $45°$, and updated at a $1\,\mathrm{ms}$ timestep, for smooth numerical integration.

To emulate real-world measurement conditions, the simulation includes a `Sensor` class that introduces two sources of error in the measured angle: instantaneous jitter and accumulated drift. The jitter represents sensor noise, while the drift term simulates sensor bias accumulation (for example, the BNO086 sensor from the physical rig claims a bias of only $0.5°$ per minute (SparkFun Electronics, 2025)). Unlike in the previous experiments, where the sensors failed, the simulation ensures that they remain operational while keeping into account their imperfections. This noise is implemented as a call to `numpy.random.normal`, which generates random values based on a Gaussian (Normal) distribution of the form $N(\mu, \sigma)$:

```
def read(self, true_angle):
    self.bias += np.random.normal(0, self.drift)
    return true_angle + self.bias + np.random.normal(0, self.jitter)
```

The control structure is implemented through a `controlStrategy` class, which supports On-Off, Proportional and PID control strategies. The PID controller computes the control signal based on incremental approximations of the integral and derivative terms, as described in the previous experiments, based on the error, still computed as $e(t) = r(t) - y(t)$. The controller gains are defined within the code and remain constant during each run. However, they can be tweaked between simulations.

Another key point in the implementation of a control system is the decoupling between the physics timestep ($1\,\mathrm{ms}$) and the control timestep ($10\,\mathrm{ms}$). The control loop only executes every 10 physics iterations, to allow for precise physical computations while maintaining realism in the speed of the controller, as real control systems cannot operate at $1\,\mathrm{kHz}$ like the simulation's physics.

The main software architecture of the simulation is structured through Python classes for the oscillator dynamics, the noisy sensor and the control algorithm. As the simulation runs, it collects timestamped values of the true angle, measured angle, and control signal of the system, saving them to an in-memory list that is then written to a CSV file when the experiment is finalized. This CSV log is similar in concept to the data files that the physical rigs were intended to generate, with the goal of analyzing them through separate plotting scripts. As such, the simulation itself does not generate graphs, but rather only the numerical data for the user to then analyze how they see fit.

Most of the code behind calculations such as those of PID control has not been included in this section, as it only differs in syntax (C++ vs Python) from that of the physical experiments. However, it is important to reiterate that, even though the code implementations share substantial similarities, the simulation does not aim to replace or fully replicate the physical rigs, as it is simplified and idealized.

3.3.2 Simulation Results and Analysis

The simulation was executed with three control strategies: On-Off, Proportional, and PID, each evaluated both with a constant setpoint as well as a changing setpoint midway through the simulation. For each case, the true angle, measured angle and target have been recorded and graphed. The following paragraphs analyze the behavior observed in each control strategy, mostly with the theoretical concepts described earlier. For all graphs, the true angle of the system is represented in blue, while the measured angle by the sensor is graphed in red. A horizontal gray dashed bar shows the system's setpoint at a certain time, and a vertical green dotted line represents the point at which the system's setpoint changes, if applicable to that specific graph.

The first control strategy evaluated was the On-Off controller, the simplest form of closed-loop control. In this mode, the control output is either its maximum or minimum depending on whether the error is positive or negative. As expected and shown in Figure 3.6, this binary response produces a continuous oscillation around the setpoint, constantly overshooting and undershooting it. When simulated, the oscillator entered a constant repeating pattern where the system's state did not converge to the setpoint. Instead, the motion remained bounded but oscillatory, classifying this system as a marginally stable system as defined in the theoretical framework. This behavior is expected of On-Off control, not diverging but not converging either.

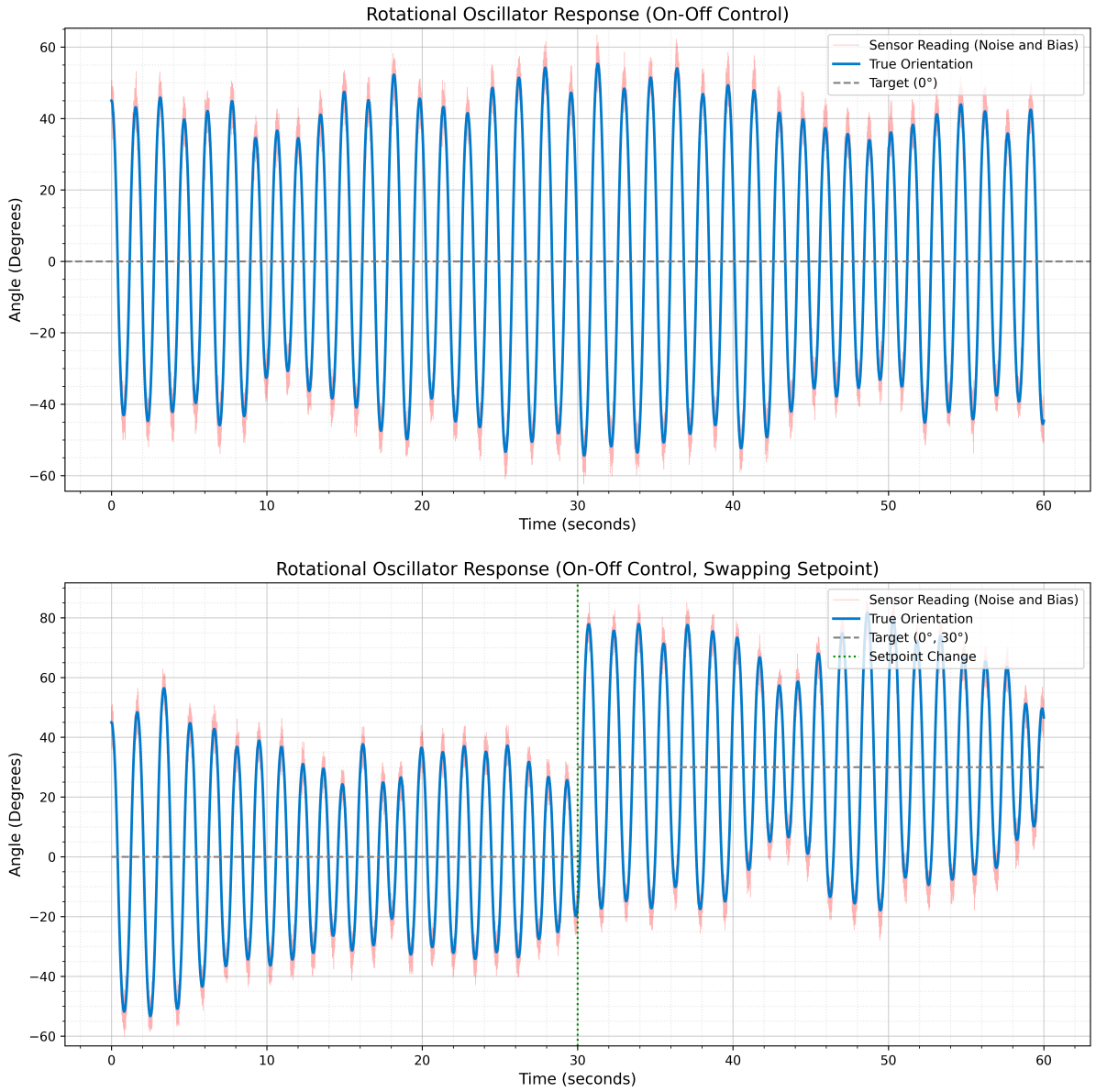


Figure 3.6: Simulation results under the On-Off control strategy. The upper graph shows a simulation with a constant setpoint, while the lower graph shows a simulation where the setpoint is changed to 30° halfway through its runtime.

The top graph of Figure 3.6 shows the system's response for a constant setpoint. Because the magnitude of the response cannot be altered, the system is unable to tune its output to approach the setpoint without exceeding it. As a result, the system does not converge and oscillations persist throughout the simulation. A steady-state error is present, but its value cannot be determined as oscillatory functions do not have a defined limit when tending to infinity. Since oscillations are always present and they do not reduce over time, a settling time cannot be defined either, since settling involves convergence to a bounded region without leaving it.

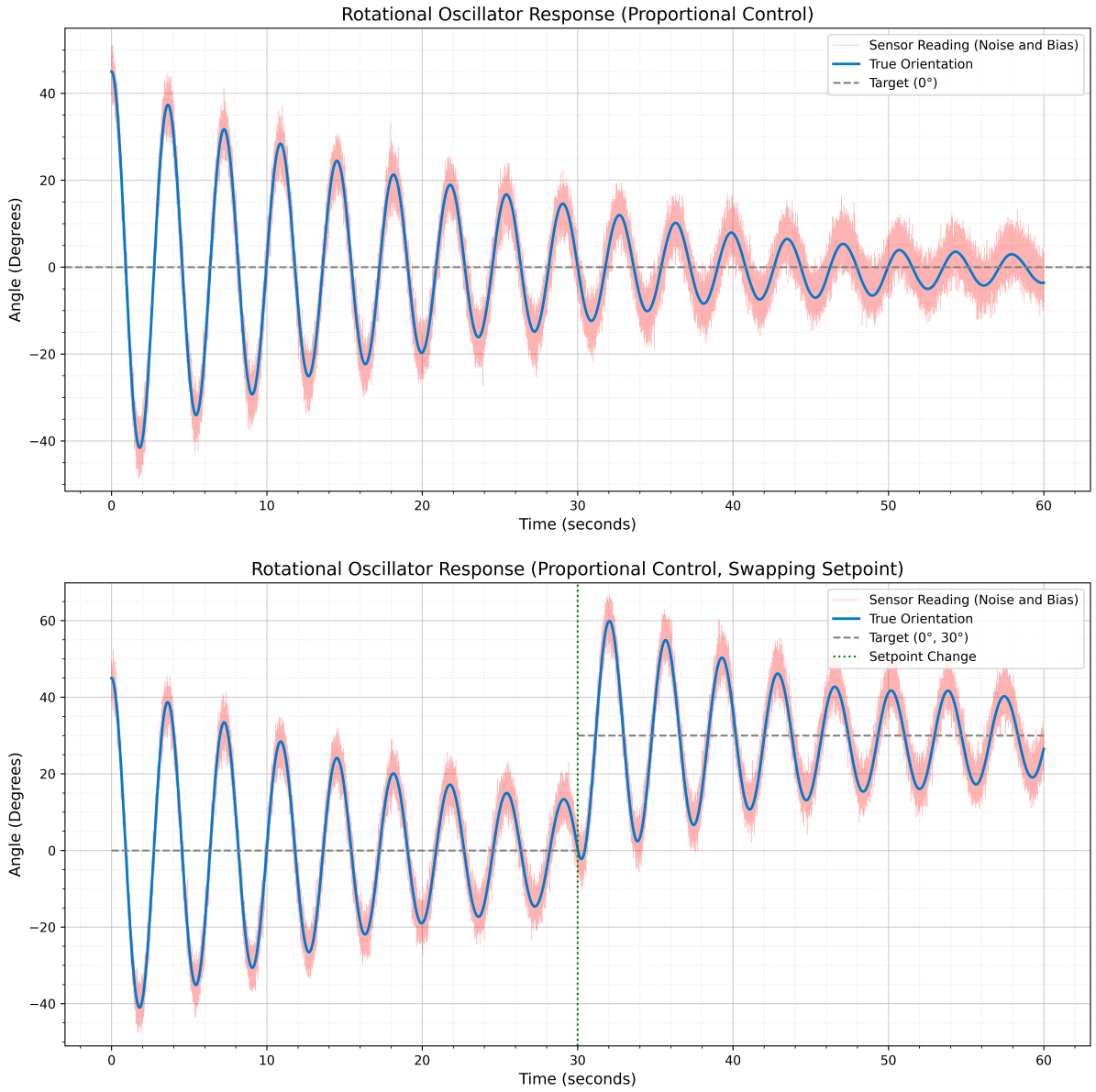


Figure 3.7: Simulation results for the proportional control strategy. The upper graph shows the simulation with a constant setpoint, while the lower graph shows the simulation where the setpoint is changed to 30° halfway through its runtime.

The bottom graph of Figure 3.6 introduces a change in setpoint halfway through the simulation. In this scenario, the controller immediately reacts to this change by applying maximum force in the appropriate direction. However, after exceeding the new setpoint, the system returned to an oscillatory behavior. Although the amplitude of the oscillations was not constant, it did not follow a consistent reducing pattern either.

The second evaluated strategy was the proportional controller, in which the control output is directly proportional to the error between the setpoint and the measured state. Unlike the discontinuous control signal of On-Off control, proportional control outputs a continuous signal with variable magnitude dependent on the distance to the setpoint. In Figure 3.7, the expected change in behavior to On-Off control is shown, outputting a smoother and more

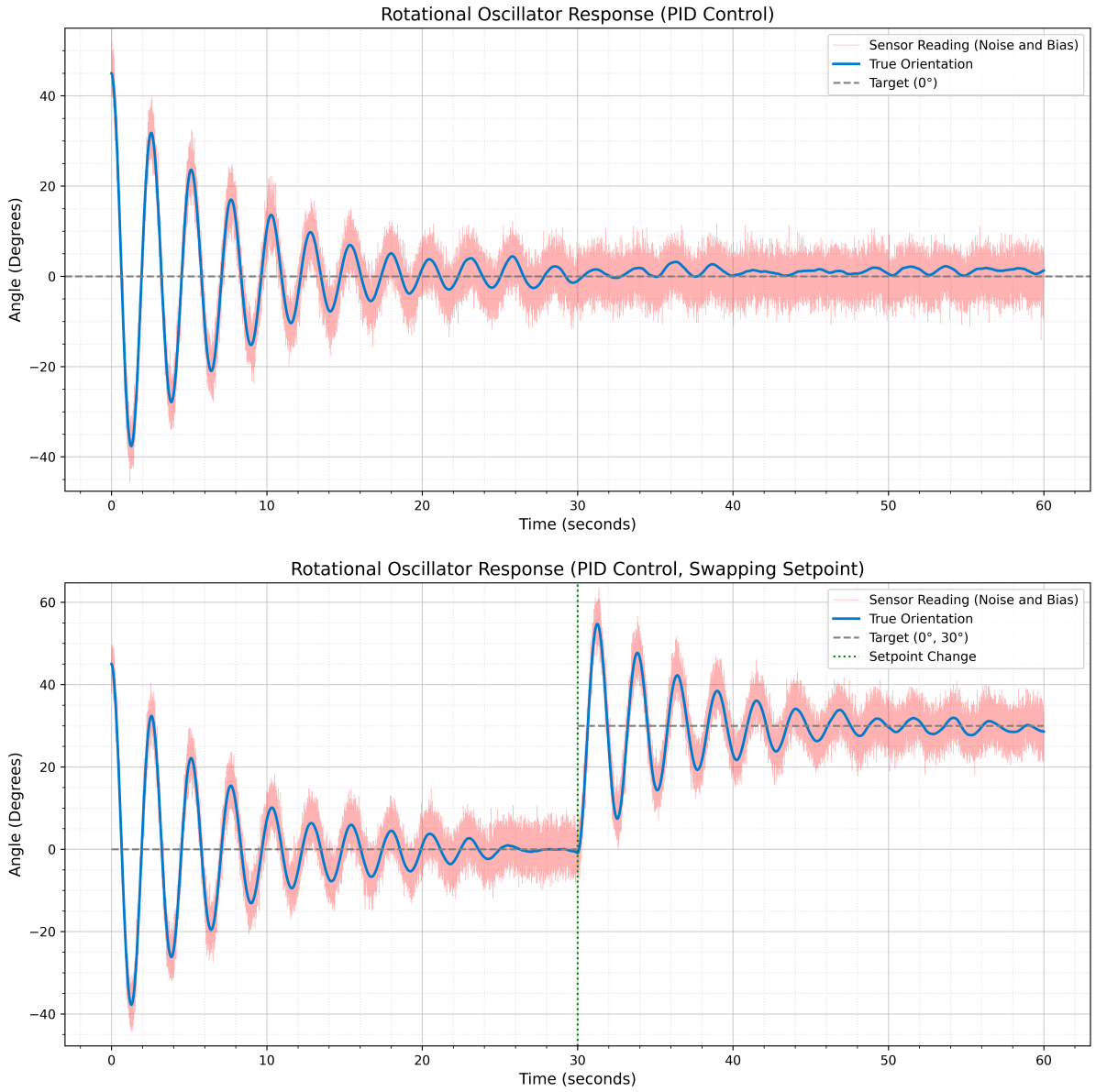


Figure 3.8: Simulation results for the proportional-integral-derivative (PID) control strategy. The upper graph shows the simulation with a constant setpoint, while the lower graph shows the simulation where the setpoint is changed to 30° halfway through its runtime.

controlled response. When under a constant setpoint, the system converged towards the setpoint. However, the system did not converge exactly to the setpoint, but rather stabilized with a non-zero steady-state error. This means that, while the system was approaching the setpoint, it did not actually settle on it.

Figure 3.7 shows this behavior clearly. In the upper graph, the true angle slowly oscillates with smaller and smaller amplitude. In the lower graph, a transient phase can be observed in the switch from one setpoint to another, with massive initial oscillations slowly decaying and converging to the setpoint.

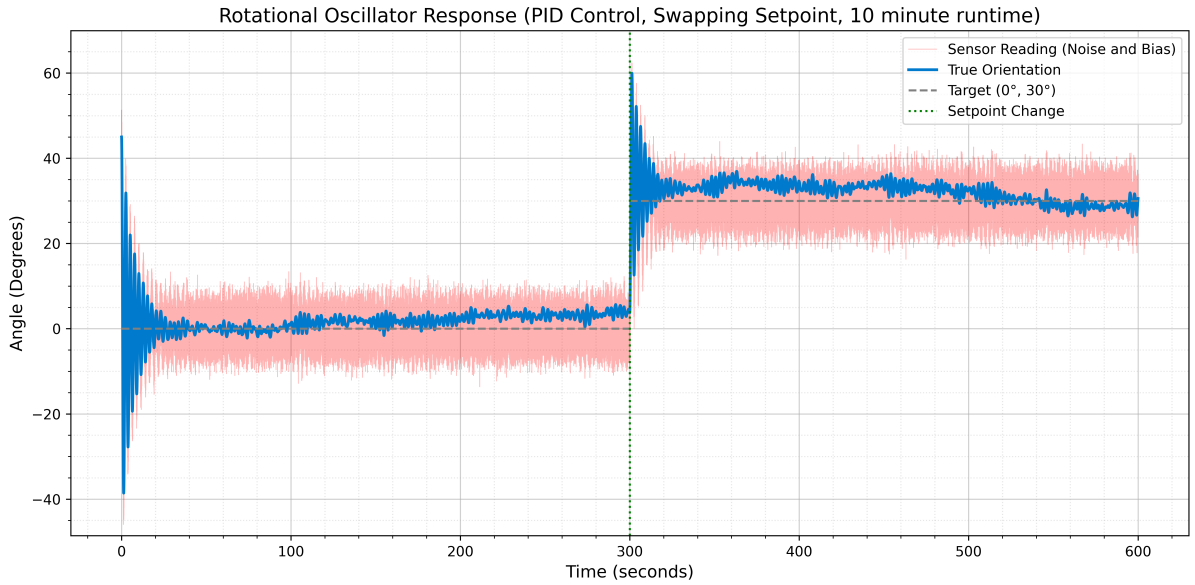


Figure 3.9: Simulation results from a PID controller with an extended runtime of ten minutes, showing how measurement bias (or sensor drift) affects steady-state error.

The third control strategy was the PID controller, which combines proportional, integral and derivative terms. As shown in Figure 3.8, the PID controller produced a substantially different response from the On-Off and proportional controllers. Namely, the controller reduced the amplitude of the transient oscillations, reduced the settling time, and also reduced the long-term error toward a much smaller value, around 3° at maximum in the constant setpoint simulation. In the changing setpoint simulation, the control loop's response shows transient behavior when the setpoint is changed. This transient behavior is shown as an initial rise, a short overshoot, and then a decaying oscillation, after which the true angle nearly converges to the target. This behavior classifies the system as asymptotically stable, as the state is bounded and approaches the setpoint as time passes. The integral term is responsible for eliminating (or at least strongly reducing) the error that remained under proportional control, while the derivative term improved damping and reduced the overshoot, shortening the settling time. Overall, the PID controller was the most successful both in terms of transient performance (rise time, overshoot and settling time) and steady-state performance (near-zero steady-state error) of the three strategies tested, at the cost of more complex tuning and a more costly implementation in terms of computational power and memory allocation.

In addition to the previous simulations, a long-duration experiment was conducted using the PID controller for ten consecutive minutes, including a setpoint change midway, to examine the system's behavior over an extended runtime and the effects of sensor drift. Figure 3.9 shows the resulting response. Both transient phases (at initialization and at the setpoint change) lasted roughly ten seconds, with characteristic rise time, overshoot and damping. However, during the remaining timespan of the simulation, measurement bias caused the controller to slightly deviate from the setpoint. In the first half of the simulation, the sensor reported lower

values than the system's real state, causing the PID controller to overshoot the setpoint, slowly deviating more and more. During the second half of the experiment, especially in the last two minutes, the sensor read higher values than those of the system's real state, causing the controller to slightly undershoot the setpoint. This simulation exemplifies the fact that, even if a system is designed and tuned to be asymptotically stable, a non-ideal sensor may still cause deviations from the desired state.

4 CONCLUSIONS

The work developed throughout this project has provided me with a meaningful understanding of how control theory governs the behavior of autonomous aerial systems. Addressing the first research objective, by building its foundation through sensors, actuators, and feedback, the project has allowed me to see how an abstract concept such as control theory and its mathematical modeling can impact a real system, and how it translates into decisions about hardware, software, and implementation. The theoretical framework provided the basis of this field of study in the form of concepts such as feedback, stability, steady-state error, or transient phase, among others, which later manifested in the practical framework, especially within the Python simulations.

As for the second objective, regarding the implementation of control strategies, studying them in detail provided a foundation for understanding how drones remain stable in flight. In analyzing On-Off control, its trade-offs and unsuitability for models requiring precise control, such as drones, could be observed, and proportional control also showed its shortcomings in the form of perpetual oscillations. By modeling a PID controller and tuning it to achieve the desired response, I gained an intuition for why drones require such complicated mathematical structures to control them rather than just using simple logic. In addition, rewriting the control logic both for C++ and Python helped me understand it more deeply, and by the time it was rewritten in Python, I noticed I had subconsciously applied many changes in implementation to streamline the computational process that would have otherwise been left unimproved.

Although the construction of both physical rigs, the rotational oscillator and the vertical oscillator, did not yield any usable experimental data due to electromagnetic interference, the process of designing, assembling, wiring, and programming them was a deeply instructive experience, effectively completing the third objective of building simplified drone subsystems. Building these experiments required making decisions about materials, design, placement, power distribution, and communication protocols, among others. Additionally, since the interference prevented the rigs from collecting stable data and thus controlling their system while the motors were active, this project also taught me a lesson in sensors' sensitivity to external interference, especially when using protocols such as I^2C. This observation of real-world constraints and sensor limitations directly answers my fourth research objective. This ultimate failure made me gain an admiration towards how complex a functioning system truly is to design, and the meticulous engineering behind commercial systems to achieve such remarkable, reliable, and stable performances.

The failure of the physical experiments was one of the main motivating forces behind the creation of the simulation. This simulated environment allowed me to complete the analysis of stability proposed in the third objective, which could not be carried out on the physical rigs. It provided a way to accurately graph the controller's various responses, as well as visualize and showcase the theoretical concepts and phenomena, such as oscillations and transient response. Through these simulations, I was able to demonstrate how an On-Off controller keeps the system bounded but cannot converge, how proportional control reduces error but inevitably oscillates around the setpoint, and how PID control stabilizes the system much more rapidly with minimal steady-state error. These behaviors matched the expectations for each controller due to their mathematical properties, bringing the mathematical descriptions of control together with their observed behavior, as proposed in my final objective.

Despite the fact that the project faced technical difficulties, I would consider the overall experience to be profoundly valuable, as it has taught me the importance of planning and organization, especially when dealing with small hardware where small oversights can cause complete failure. The project also exposed me to the necessity of troubleshooting and implementing a system piece-by-piece rather than attempting to integrate it all at once, as well as helping me recognize when a change in approach is necessary, without falling for the "sunk-cost" fallacy in ways such as attempting to fix the physical rigs once it was clear that they could not be protected from interference.

Looking forward, there are several directions in which to improve this work. For example, a resistance to electromagnetic interference could be created by shielding cables or designing a custom printed circuit board (PCB). Additionally, filtering algorithms such as a Kalman filter could be included to improve stability by combining sensor readings and state estimations.

In conclusion, this research has allowed me to explore control theory from both the theoretical and practical perspectives, both gaining a conceptual and applied understanding of its principles. Despite the limitations of the project, the combination of theoretical study, physical construction, and simulation has provided me with a comprehensive understanding of how drones are stabilized and why control theory is so essential for autonomous flight. The project has been both technically demanding as well as personally intriguing, and it has deepened my interest in pursuing further experience in this field.

BIBLIOGRAPHY

09glasgow09. (2009, August 31). *English: Animation of Riemann sum for* $y = x^2$. Retrieved November 22, 2025, from https://commons.wikimedia.org/wiki/File:Riemann_sum_(y%3Dx%5E2).gif

Adafruit Industries. (2025, December 2). *Adafruit/adafruit_vl53l0x* (Version 1.0.0). Retrieved December 7, 2025, from https://github.com/adafruit/Adafruit_VL53L0X

Analog Devices. (2016, March 1). *MEMS gyroscope provides precision inertial sensing in harsh, high temperature environments*. Retrieved May 28, 2025, from https://www.analog.com/en/resources/technical-articles/mems-gyroscope-provides-precision-inertial-sensing.html

ArduPilot Dev Team. (2025, February 9). *What is a MultiCopter and how does it work?* ArduPilot. Retrieved February 9, 2025, from https://ardupilot.org/copter/docs/what-is-a-multicopter-and-how-does-it-work.html

Avnet Abacus. (2025, August 20). *Barometric pressure sensors*. Retrieved August 20, 2025, from https://my.avnet.com/abacus/solutions/technologies/sensors/pressure-sensors/media-types/barometric/

Bayisa, A. T., & Geng, L.-H. (2019). Controlling quadcopter altitude using PID-control system. *International Journal of Engineering Research and Technology*, *8*(12), 257–261. https://doi.org/10.17577/IJERTV8IS120118

Betaflight Dev Team. (2025, August 20). *Barometer*. Betaflight. Retrieved August 20, 2025, from https://betaflight.com/docs/wiki/guides/current/Barometer

Bold City Heating & Air. (2025, June 6). *On off control*. Retrieved September 9, 2025, from https://boldcityac.com/on-off-control/

Chapra, S. C., & Canale, R. P. (2021). *Numerical methods for engineers* (Eighth edition). McGraw-Hill.

Control Station. (2015, April 23). *What is on-off control? what is its role in industrial automation?* Retrieved September 9, 2025, from https://controlstation.com/what-is-on-off-control/

DJI. (2025, August 20). *DJI Mavic 3 Pro - specs*. Retrieved August 20, 2025, from https://www.dji.com/es/mavic-3-pro/specs

Duckietown. (2025, September 16). *PID controllers generalities*. Retrieved September 16, 2025, from https://docs.duckietown.com/ente/course-intro-to-drones/pid-controllers/theory/pid-generalities.html

Electronics For You. (2023, January 4). *Control system definition, types, applications, and FAQs*. Retrieved August 29, 2025, from https://www.electronicsforu.com/technology-trends/learn-electronics/control-system-definition-types-applications-and-faqs

Elmenreich, W. (2002). *Sensor fusion - an overview*. Retrieved August 26, 2025, from https://www.sciencedirect.com/topics/engineering/sensor-fusion

European Space Agency. (2025, August 21). *What is Galileo?* Retrieved August 21, 2025, from https://www.esa.int/Applications/Satellite_navigation/Galileo/What_is_Galileo

Evans, P. (2023, January 24). *HVACR dead band basics*. Retrieved September 9, 2025, from https://theengineeringmindset.com/hvacr-dead-band-basics/

Fisher, C. J. (2010). *AN-1057: Using an Accelerometer for inclination sensing*. Analog Devices. Retrieved August 20, 2025, from https://www.analog.com/en/resources/app-notes/an-1057.html

Franklin, G. F., Powell, J. D., & Emami-Naeini, A. (2019). *Feedback control of dynamic systems* (8th ed.). Pearson. Retrieved September 9, 2025, from https://mrce.in/ebooks/Feedback%20Control%20of%20Dynamic%20Systems%208th%20Ed.pdf

GIS Geography. (2018, April 12). *How GPS receivers work - trilateration vs triangulation*. Retrieved August 21, 2025, from https://gisgeography.com/trilateration-triangulation-gps/

Google. (2025). *Gemini* (Version 1.5 Pro) [Large language model. See Appendix for the specific prompt used.]. Retrieved July 18, 2025, from https://gemini.google.com

Haber, A. (2024, January 27). *Limitations of proportional control – why we need an integrator in the control loop*. Retrieved September 10, 2025, from https://aleksandarhaber.com/limitations-of-proportional-control-why-we-need-an-integrator-in-the-control-loop-control-engineering-lecture/

Infineon Technologies. (2020). *Infineon DPS310/368 pressure Sensor2Go kit user manual*. Version 1.0. Retrieved March 11, 2025, from https://www.mouser.es/datasheet/3/70/1/Infineon-DPS310_368_Pressure_Sensor2Go_Kit-UserManual-v01_00-EN.pdf

Infineon Technologies. (2025). *Infineon/arduino-xensiv-dps3xx* (Version 1.0.0). Retrieved May 10, 2025, from https://github.com/Infineon/arduino-xensiv-dps3xx

JHEMCU. (2025, February 13). *JHEMCU brushless wing dual 40a 2-in-1 ESC* [Product specification page]. Retrieved February 13, 2025, from https://www.jhemcu.com/e_productshow/?81-JHEMCU-BRUSELESS-WING-DUAL-40A-2IN1-ESC-81.html

Keim, R. (2020). *Introduction to capacitive accelerometers: Measuring acceleration with capacitive sensing*. All About Circuits. Retrieved August 20, 2025, from https://www.allaboutcircuits.com/technical-articles/introduction-to-capacitive-accelerometer-measure-acceleration-capacitive-sensing/

Lisleapex Electronics. (2024). *Difference between open loop and closed loop control system 2024*. Lisleapex Electronics. Retrieved September 1, 2025, from https://www.lisleapex.com/blog-difference-between-open-loop-and-closed-loop-control-system-2024

MEMS Exchange. (2025, June 19). *Lithography*. Retrieved June 19, 2025, from https://www.mems-exchange.org/MEMS/processes/lithography.html

Monolithic Power Systems. (2025, September 9). *Response time, accuracy, and stability*. Retrieved September 9, 2025, from https://www.monolithicpower.com/en/learning/mpscholar/analog-vs-digital-control/comparative-analysis/response-time-accuracy-stability?srsltid=AfmBOopVE-o2Vt5UbS0fRfOT9wL69IVIFUczZfJ4JN4tq86TzgselTij

National Coordination Office for Space-Based Positioning, Navigation, and Timing. (2025a, August 20). *Control segment*. GPS.gov. Retrieved August 20, 2025, from https://www.gps.gov/systems/gps/control/

National Coordination Office for Space-Based Positioning, Navigation, and Timing. (2025b, August 20). *GPS overview*. GPS.gov. Retrieved August 20, 2025, from https://www.gps.gov/systems/gps/

National Coordination Office for Space-Based Positioning, Navigation, and Timing. (2025c, August 20). *Space segment*. GPS.gov. Retrieved August 20, 2025, from https://www.gps.gov/systems/gps/space/

National Coordination Office for Space-Based Positioning, Navigation, and Timing. (2025d, August 21). *GPS accuracy*. GPS.gov. Retrieved August 21, 2025, from https://www.gps.gov/systems/gps/performance/accuracy/

National Instruments. (2020). *The PID controller & theory explained*. Retrieved September 16, 2025, from https://www.ni.com/en/shop/labview/pid-theory-explained.html

Nedelkovski, D. (2015, November 19). *What is MEMS? accelerometer, gyroscope & magnetometer with Arduino*. How To Mechatronics. Retrieved June 4, 2025, from https://howtomechatronics.com/how-it-works/electrical-engineering/mems-accelerometer-gyrocope-magnetometer-arduino/

Passaro, V. M. N., Cuccovillo, A., Vaiani, L., De Carlo, M., & Campanella, C. E. (2017). Gyroscope technology and applications: A review in the industrial perspective. *Sensors*, *17*(10), 2284. https://doi.org/10.3390/s17102284

RT Engineering. (2023, August 9). *Open-loop vs closed-loop control systems: Features, examples, and applications*. RT Engineering. Retrieved September 1, 2025, from https://www.rteng.com/blog/open-loop-vs-closed-loop-control-systems

Sayed, E. (2024, August 2). *Drones and PID control: An introductory guide to aerial robotics*. Retrieved September 10, 2025, from https://medium.com/@sayedebad.777/drones-and-pid-control-an-introductory-guide-to-aerial-robotics-9cf24ffb1853

Simrock, S. (2011). Tutorial on control theory. *Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2011)*. Retrieved August 29, 2025, from https://cds.cern.ch/record/1100534/files/p73.pdf

Sodhi, R. S., Kokje, S. S., & Sawant, P. P. (2024). Comparative study of microcontroller: Arduino Uno, Raspberry Pi 4, and ESP32. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, *12*(7). Retrieved September 15, 2025, from https://www.ijraset.com/best-journal/comparative-study-of-microcontroller-esp32-arduino-uno-raspberry-pi-4

SparkFun Electronics. (2025, December 7). *SparkFun VR IMU breakout - BNO086 (Qwiic)*. Retrieved December 7, 2025, from https://www.sparkfun.com/sparkfun-vr-imu-breakout-bno086-qwiic.html

Suministros Miguel Lopez. (2025, August 14). *Barra cromada ø 16 mm f114 x 1mtr*. Retrieved August 14, 2025, from https://www.suministrosmiguellopez.es/producto/barra-cromada-16-mm-f114-x-1mtr-cr16

Texas Instruments. (2021). *What is a Hall-effect sensor?* (Tech. rep. No. SSZT164). Texas Instruments. Retrieved September 25, 2025, from https://www.ti.com/lit/ta/sszt164/sszt164.pdf?ts=1765348690788

Thingiverse Contributor. (2025, December 10). Customizable linear bushing / bearing [stl file] [Original file no longer accessible. Previously hosted on Thingiverse]. Retrieved December 10, 2025, from https://www.thingiverse.com

Toa, M., & Whitehead, A. (2019). *Ultrasonic sensing basics*. Texas Instruments. https://www.ti.com/lit/an/slau720a/slau720a.pdf

Tutorialspoint. (2025, September 9). *Control systems - steady state errors*. Retrieved September 9, 2025, from https://www.tutorialspoint.com/control_systems/control_systems_steady_state_errors.htm

UAV Navigation. (2025, August 4). *Magnetometer, why is it critical for UAV navigation?* Retrieved September 25, 2025, from https://www.uavnavigation.com/company/blog/uav-navigation-depth-magnetometer

VectorNav. (2025a, May 21). *Learn about MEMS accelerometers, gyroscopes, and magnetometers*. VectorNav. Retrieved May 21, 2025, from https://www.vectornav.com/resources/inertial-navigation-primer/theory-of-operation/theory-mems

VectorNav. (2025b, September 25). *Learn how an attitude and heading reference system (AHRS) works*. Retrieved September 25, 2025, from https://www.vectornav.com/resources/inertial-navigation-primer/theory-of-operation/theory-ahrs

Wikipedia Contributors. (2025, January 8). Aircraft principal axes. In *Wikipedia*. Retrieved February 14, 2025, from https://en.wikipedia.org/w/index.php?title=Aircraft_principal_axes&oldid=1268147601

WPILib. (2025, September 9). *Bang-bang control with BangBangController*. FIRST Robotics Competition. Retrieved September 9, 2025, from https://docs.wpilib.org/en/stable/docs/software/advanced-controls/controllers/bang-bang.html

x-engineer.org. (2025, September 10). *Proportional (p) controller*. Retrieved September 10, 2025, from https://x-engineer.org/proportional-controller/

Zheng, L., Yang, R., Pan, J., & Cheng, H. (2021, April 21). Safe learning-based tracking control for quadrotors under wind disturbances. https://doi.org/10.48550/arXiv.2009.01992

5 APPENDIX

This appendix compiles additional images for the physical experiments, as well as the full code for both physical experiments and the Python simulation.

5.1 PHYSICAL RIG IMAGES

This section consists of images of both physical rigs as well as my workspace while creating them.

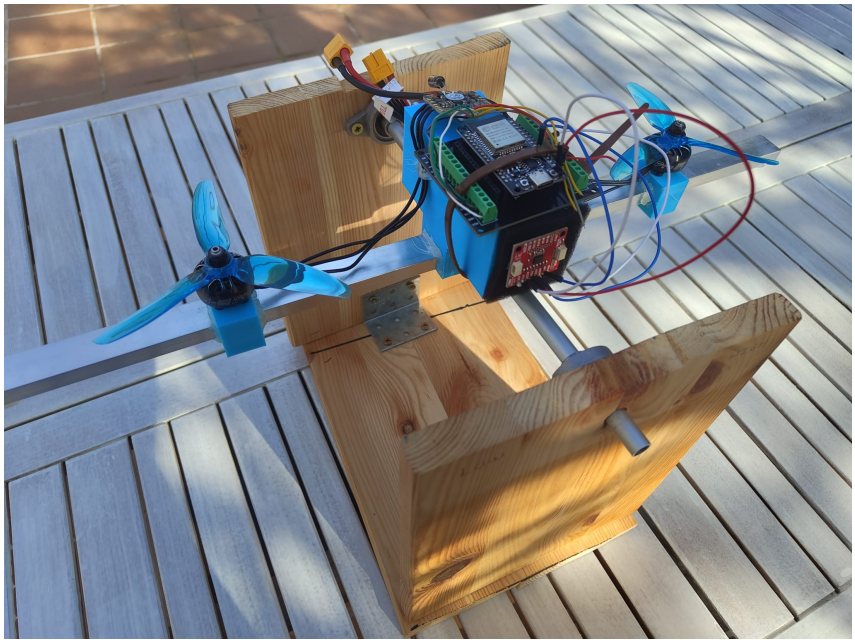


Figure 5.1: Posterior view of the rotational oscillator experiment (Own Photograph).

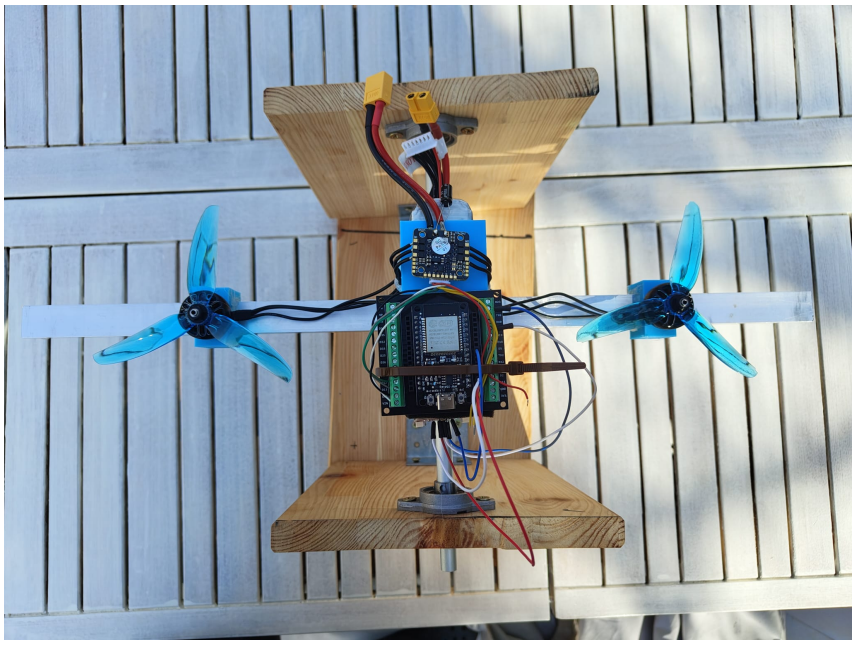


Figure 5.2: Top view of the rotational oscillator experiment (Own Photograph).

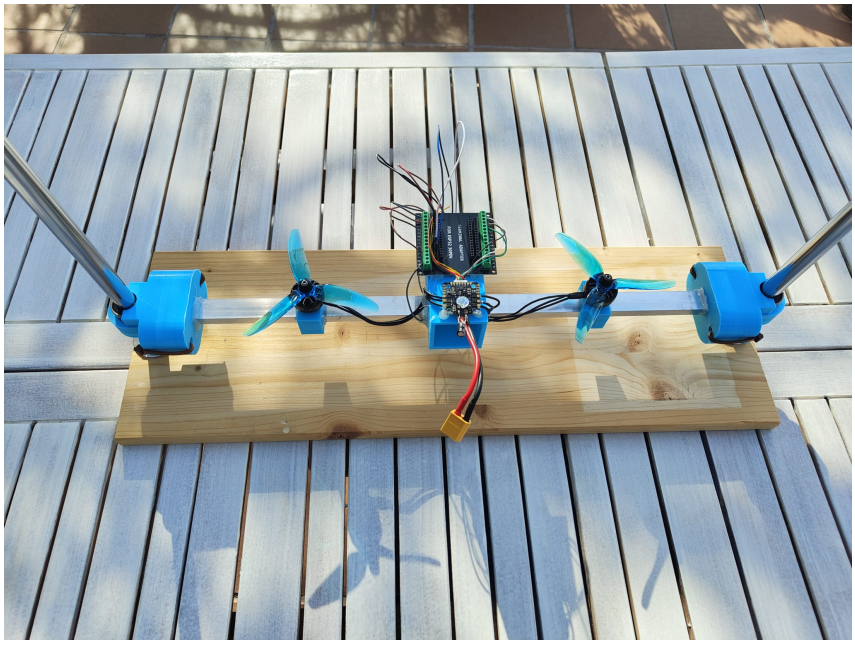


Figure 5.3: Frontal view of the vertical oscillator experiment (Own Photograph).

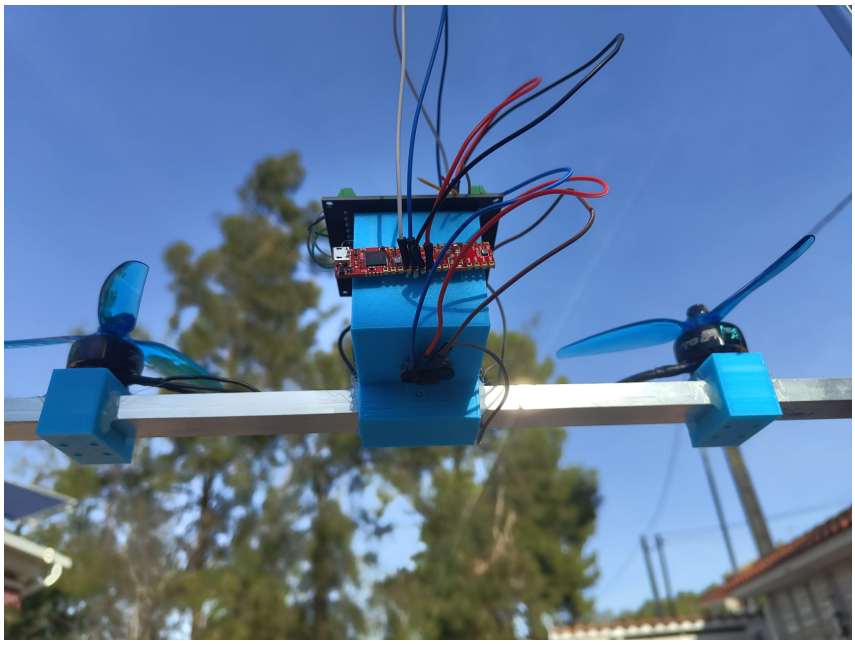


Figure 5.4: Bottom view of the vertical oscillator experiment, with the TOF sensor in view as well as the barometric altimeter (Own Photograph).

Figure 5.5: Main workspace for the design and creation of the physical rigs (Own Photograph).

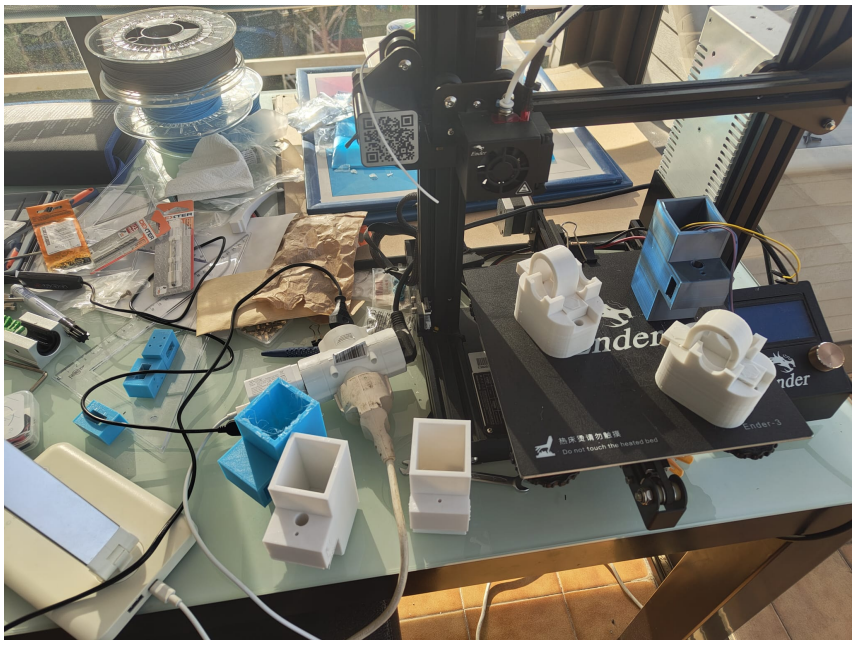


Figure 5.6: Lateral side of my workspace with initial 3D printed prototypes (Own Photograph).

5.2 ROTATIONAL OSCILLATOR CODE

This section compiles all of the code used for the rotational oscillator.

`main.cpp`:

```cpp
#include <Arduino.h>
#include <vector>
#include <Wire.h>
#include "SparkFun_BNO08x_Arduino_Library.h"

// Web functionality declarations
void initWebServer();
void handleWebServer();

// Gyro
BNO08x gyro;
float lastPitch = 0.0;
unsigned long lastSensorUpdateTime = 0;

// CONSTANT DECLARATIONS
constexpr int MOTOR1 = 13;
constexpr int MOTOR2 = 12;
constexpr int PWM_CH1 = 0;
constexpr int PWM_CH2 = 1;
constexpr int PWM_FREQ = 50; // Hz control signal
constexpr int PWM_RESOLUTION = 16; // resolution in bits
```

```
22
23 float gain_p = 0.2;
24 float gain_d = 0.05;
25 float gain_i = 0.1;
26 unsigned long lastLoopTime = 0;
27 float integralSum = 0.0;
28 float error = 0.0;
29 float lastError = 0.0;
30
31 // Calibration Variables
32 float calibMiddle = 0.0;
33 float calibLowVal = 0.0;
34 float calibHighVal = 0.0;
35
36 struct DataPoint {
37     uint32_t timestamp;
38     float sensorValue;
39     int selectedSensor;
40     float error;
41     float controlOutput;
42     int strategyUsed;
43 };
44
45 volatile bool running = false;
46 volatile int selectedStrategy = 0;
47 volatile int selectedSensor = 0;
48 bool wasRunning = false;
49
50 // DATA BUFFER (Used for streaming)
51 std::vector<DataPoint> dataLog;
52
53 // ESC range (microseconds)
54 constexpr uint32_t ESC_MIN_PULSE = 1000;
55 constexpr uint32_t ESC_MAX_PULSE = 2000;
56
57 // Control strategy output range
58 constexpr float STRATEGY_OUTPUT_MIN = 0.0;
59 constexpr float STRATEGY_OUTPUT_MAX = 1.0;
60
61 // Derived constants
62 constexpr float PERIOD_US = 1000000.0f / PWM_FREQ;
63 constexpr uint32_t MAX_DUTY = (1UL << PWM_RESOLUTION) - 1UL;
64 constexpr uint32_t MIN_DUTY = (uint32_t)((ESC_MIN_PULSE / PERIOD_US) *
   ↪  (float)MAX_DUTY + 0.5f);
65
66 // FUNCTIONS
```

```
67
68 float clamp(float v, float a, float b) {
69   if (v < a) return a;
70   if (v > b) return b;
71   return v;
72 }
73
74 float mapFloat(float x, float in_min, float in_max, float out_min, float
   ↪  out_max) {
75   float t = (x - in_min) / (in_max - in_min);
76   return out_min + t * (out_max - out_min);
77 }
78
79 uint32_t controlToDuty(float controlValue) {
80   float v = clamp(controlValue, STRATEGY_OUTPUT_MIN,
     ↪  STRATEGY_OUTPUT_MAX);
81   float pulse = mapFloat(v, STRATEGY_OUTPUT_MIN, STRATEGY_OUTPUT_MAX,
     ↪  ESC_MIN_PULSE, ESC_MAX_PULSE);
82   float fraction = pulse / PERIOD_US;
83   fraction = clamp(fraction, 0.0f, 1.0f);
84   uint32_t duty = (uint32_t)(fraction * (float)MAX_DUTY + 0.5f);
85   return duty;
86 }
87
88 void setMotorPower(float controlValue) {
89   if (controlValue > 0) {
90     uint32_t duty = controlToDuty(controlValue);
91     ledcWrite(PWM_CH1, duty);
92     ledcWrite(PWM_CH2, MIN_DUTY);
93   }
94   else if (controlValue < 0) {
95     uint32_t duty = controlToDuty(-controlValue);
96     ledcWrite(PWM_CH1, MIN_DUTY);
97     ledcWrite(PWM_CH2, duty);
98   }
99   else {
100    ledcWrite(PWM_CH1, MIN_DUTY);
101    ledcWrite(PWM_CH2, MIN_DUTY);
102  }
103 }
104
105 float sensorRead(int selectedSensor) {
106   for(int i = 0; i < 5; i++) {
107     if (gyro.getSensorEvent() && gyro.getSensorEventID() ==
       ↪  SENSOR_REPORTID_GAME_ROTATION_VECTOR) {
108       lastSensorUpdateTime = millis();
```

```
109       lastPitch = gyro.getPitch();
110     }
111     delay(2);
112   }
113   return lastPitch;
114 }
115
116 void calibrateLowStep(int selectedSensor) {
117     float sum = 0.0;
118     for(int i = 0; i < 200; i++) { sum += sensorRead(selectedSensor);
        ↪  delay(10); }
119     calibLowVal = sum / 200.0;
120     calibMiddle = (calibLowVal + calibHighVal) / 2.0;
121 }
122
123 void calibrateHighStep(int selectedSensor) {
124     float sum = 0.0;
125     for(int i = 0; i < 200; i++) { sum += sensorRead(selectedSensor);
        ↪  delay(10); }
126     calibHighVal = sum / 200.0;
127     calibMiddle = (calibLowVal + calibHighVal) / 2.0;
128 }
129
130 float runControl(float sensorValue, int selectedStrategy) {
131   unsigned long now = micros();
132   float dt = (now - lastLoopTime) / 1000000.0;
133   lastLoopTime = now;
134   lastError = error;
135   error = calibMiddle - sensorValue;
136
137   switch(selectedStrategy) {
138     default: return 0.0;
139     case 1: return gain_p * error;
140     case 2: {
141       if (error > 0) return 1.0;
142       else if (error < 0) return -1.0;
143       else return 0.0;
144     }
145     case 3: {
146       // Prevent Integral Windup
147       integralSum += error * dt;
148       float derivative = (error - lastError) / dt;
149       return gain_p * error + gain_i * integralSum + gain_d *
          ↪  derivative;
150     }
151   }
```

```
152 }
153
154 // Data streaming helper (too large to keep in esp32 memory)
155 String getBufferCSV() {
156     String csv = "";
157     if (dataLog.empty()) return "";
158
159     // Convert buffer to string
160     for (const auto& dp : dataLog) {
161         csv += String(dp.timestamp) + "," +
162                String(dp.sensorValue, 9) + "," +
163                String(dp.selectedSensor) + "," +
164                String(dp.error, 9) + "," +
165                String(dp.controlOutput, 9) + "," +
166                String(dp.strategyUsed) + "\n";
167     }
168     // Clear the buffer so it can fill up again
169     dataLog.clear();
170     return csv;
171 }
172
173 // SETUP AND LOOP
174
175 void setup() {
176   delay(7000);
177   Serial.begin(115200);
178   Serial.println("Booting ESP32...");
179
180   dataLog.reserve(1500); // Reserve memory
181
182   Wire.begin(21, 22);
183   Wire.setClock(100000); // 100kHz
184
185   gyro.begin(0x4B, Wire, -1, -1);
186   delay(500);
187
188   gyro.enableGameRotationVector(20);
189
190   ledcSetup(PWM_CH1, PWM_FREQ, PWM_RESOLUTION);
191   ledcSetup(PWM_CH2, PWM_FREQ, PWM_RESOLUTION);
192   ledcAttachPin(MOTOR1, PWM_CH1);
193   ledcAttachPin(MOTOR2, PWM_CH2);
194
195   Serial.println("Starting web server...");
196   initWebServer();
197   Serial.println("Web server started.");
```

```
198
199   ledcWrite(PWM_CH1, MIN_DUTY);
200   ledcWrite(PWM_CH2, MIN_DUTY);
201   delay(2000);
202 }
203
204 void loop() {
205   handleWebServer();
206   if (gyro.getSensorEvent()) {
207       if (gyro.getSensorEventID() ==
          ↪  SENSOR_REPORTID_GAME_ROTATION_VECTOR) {
208         lastSensorUpdateTime = millis();
209         lastPitch = gyro.getPitch();
210       }
211   }
212
213   if (running && (millis() - lastSensorUpdateTime) < 100) {
214     if (!wasRunning) {
215       dataLog.clear();
216       wasRunning = true;
217       integralSum = 0.0;
218       lastLoopTime = micros();
219     }
220
221     int currentSensor = selectedSensor;
222     int currentStrategy = selectedStrategy;
223
224     float controlOutput = runControl(lastPitch, currentStrategy);
225     setMotorPower(controlOutput);
226
227     if (dataLog.size() < 1500) {
228         DataPoint dp;
229         dp.timestamp = micros();
230         dp.sensorValue = lastPitch;
231         dp.selectedSensor = selectedSensor;
232         dp.error = error;
233         dp.controlOutput = controlOutput;
234         dp.strategyUsed = selectedStrategy;
235         dataLog.push_back(dp);
236     }
237   }
238   else {
239     setMotorPower(0.0);
240     if (wasRunning) wasRunning = false;
241   }
242 }
```

web.cpp has been omitted from this section to avoid unnecessary stretching of the research paper and due to the fact that it is virtually identical to the vertical oscillator's web.cpp.

5.3 VERTICAL OSCILLATOR CODE

This section compiles all of the code used for the vertical oscillator.

main.cpp:

```cpp
#include <Arduino.h>
#include <vector>
#include <Wire.h>
#include <Dps3xx.h>
#include <Adafruit_VL53L0X.h>

// Web functionality declarations
void initWebServer();
void handleWebServer();

// SENSOR OBJECTS
Dps3xx dps;
Adafruit_VL53L0X lox = Adafruit_VL53L0X();

// CONSTANT DECLARATIONS
constexpr int MOTOR1 = 13;
constexpr int MOTOR2 = 12;
constexpr int PWM_CH1 = 0;
constexpr int PWM_CH2 = 1;
constexpr int PWM_FREQ = 50; // Hz control signal
constexpr int PWM_RESOLUTION = 16; // resolution in bits

// Tweak later
float gain_p = 0.2;
float gain_d = 0.05;
float gain_i = 0.1;
unsigned long lastLoopTime = 0;
float integralSum = 0.0;
float error = 0.0;
float lastError = 0.0;

// Calibration Variables
float calibMiddle = 0.0;
float calibLowVal = 0.0;
float calibHighVal = 0.0;

struct DataPoint {
```

```
38      uint32_t timestamp;
39      float sensorValue;
40      int selectedSensor;
41      float error;
42      float controlOutput;
43      int strategyUsed;
44  };
45
46  volatile bool running = false;
47  volatile int selectedStrategy = 0;
48  volatile int selectedSensor = 0;
49  bool wasRunning = false;
50
51  // DATA BUFFER (Used for streaming)
52  std::vector<DataPoint> dataLog;
53
54  // ESC range (microseconds)
55  constexpr uint32_t ESC_MIN_PULSE = 1000;
56  constexpr uint32_t ESC_MAX_PULSE = 2000;
57
58  // Control strategy output range
59  constexpr float STRATEGY_OUTPUT_MIN = 0.0;
60  constexpr float STRATEGY_OUTPUT_MAX = 1.0;
61
62  // Derived constants
63  constexpr float PERIOD_US = 1000000.0f / PWM_FREQ;
64  constexpr uint32_t MAX_DUTY = (1UL << PWM_RESOLUTION) - 1UL;
65  constexpr uint32_t MIN_DUTY = (uint32_t)((ESC_MIN_PULSE / PERIOD_US) *
    ↪  (float)MAX_DUTY + 0.5f);
66
67  // FUNCTIONS
68
69  float clamp(float v, float a, float b) {
70    if (v < a) return a;
71    if (v > b) return b;
72    return v;
73  }
74
75  float mapFloat(float x, float in_min, float in_max, float out_min, float
    ↪  out_max) {
76    float t = (x - in_min) / (in_max - in_min);
77    return out_min + t * (out_max - out_min);
78  }
79
80  uint32_t controlToDuty(float controlValue) {
```

```
  float v = clamp(controlValue, STRATEGY_OUTPUT_MIN,
  ↪  STRATEGY_OUTPUT_MAX);
  float pulse = mapFloat(v, STRATEGY_OUTPUT_MIN, STRATEGY_OUTPUT_MAX,
  ↪  ESC_MIN_PULSE, ESC_MAX_PULSE);
  float fraction = pulse / PERIOD_US;
  fraction = clamp(fraction, 0.0f, 1.0f);
  uint32_t duty = (uint32_t)(fraction * (float)MAX_DUTY + 0.5f);
  return duty;
}

void setMotorPower(float controlValue) {
  if (controlValue > 0) {
    uint32_t duty = controlToDuty(controlValue);
    ledcWrite(PWM_CH1, duty);
    ledcWrite(PWM_CH2, duty);
  }
  else {
    ledcWrite(PWM_CH1, MIN_DUTY);
    ledcWrite(PWM_CH2, MIN_DUTY);
  }
}

float sensorRead(int selectedSensor) {
  float val = 0.0;

  if (selectedSensor == 1) {
    // 7 is default oversampling rate for standard precision (2^7
    ↪  samples)
    dps.measurePressureOnce(val, 7);
  }
  else if (selectedSensor == 2) {
    VL53L0X_RangingMeasurementData_t measure;
    lox.rangingTest(&measure, false);
    if (measure.RangeStatus != 4) {  // not failed
      val = (float)measure.RangeMilliMeter;
    } else {
      val = 0.0; // Error or out of range
    }
  }
  return val;
}

void calibrateLowStep(int selectedSensor) {
    float sum = 0.0;
    for(int i = 0; i < 200; i++) { sum += sensorRead(selectedSensor);
    ↪  delay(10); }
```

```
123     calibLowVal = sum / 200.0;
124     calibMiddle = (calibLowVal + calibHighVal) / 2.0;
125 }
126
127 void calibrateHighStep(int selectedSensor) {
128     float sum = 0.0;
129     for(int i = 0; i < 200; i++) { sum += sensorRead(selectedSensor);
        ↪  delay(10); }
130     calibHighVal = sum / 200.0;
131     calibMiddle = (calibLowVal + calibHighVal) / 2.0;
132 }
133
134 float runControl(float sensorValue, int selectedStrategy) {
135   unsigned long now = micros();
136   float dt = (now - lastLoopTime) / 1000000.0;
137   lastLoopTime = now;
138   lastError = error;
139   error = calibMiddle - sensorValue;
140
141   switch(selectedStrategy) {
142     default: return 0.0;
143     case 1: return gain_p * error;
144     case 2: {
145       if (error > 0) return 1.0;
146       else if (error < 0) return -1.0;
147       else return 0.0;
148     }
149     case 3: {
150       integralSum += error * dt;
151       float derivative = (error - lastError) / dt;
152       return gain_p * error + gain_i * integralSum + gain_d *
          ↪  derivative;
153     }
154   }
155 }
156
157 // Data streaming helper (too large to keep in esp32 memory)
158 String getBufferCSV() {
159     String csv = "";
160     if (dataLog.empty()) return "";
161
162     // Convert buffer to string
163     for (const auto& dp : dataLog) {
164         csv += String(dp.timestamp) + "," +
165                String(dp.sensorValue, 9) + "," +
166                String(dp.selectedSensor) + "," +
```

```
                String(dp.error, 9) + "," +
                String(dp.controlOutput, 9) + "," +
                String(dp.strategyUsed) + "\n";
    }
    // Clear the buffer so it can fill up again
    dataLog.clear();
    return csv;
}

// SETUP AND LOOP

void setup() {
  delay(7000);
  Serial.begin(115200);
  Serial.println("Booting ESP32...");

  dataLog.reserve(1500); // Reserve memory

  ledcSetup(PWM_CH1, PWM_FREQ, PWM_RESOLUTION);
  ledcSetup(PWM_CH2, PWM_FREQ, PWM_RESOLUTION);
  ledcAttachPin(MOTOR1, PWM_CH1);
  ledcAttachPin(MOTOR2, PWM_CH2);

  // Initialize Sensors
  Wire.begin();
  dps.begin(Wire); // DPS368
  if (!lox.begin()) {
    Serial.println(F("Failed to boot VL53L0X"));
  }

  Serial.println("Starting web server...");
  initWebServer();
  Serial.println("Web server started.");

  ledcWrite(PWM_CH1, MIN_DUTY);
  ledcWrite(PWM_CH2, MIN_DUTY);
  delay(2000);
}

void loop() {
  handleWebServer();
  if (running) {
    if (!wasRunning) {
      dataLog.clear();
      wasRunning = true;
      integralSum = 0.0;
```

```
213       lastLoopTime = micros();
214     }
215
216     int currentSensor = selectedSensor;
217     int currentStrategy = selectedStrategy;
218
219     float sensorValue = sensorRead(currentSensor);
220
221     float controlOutput = runControl(sensorValue, currentStrategy);
222     setMotorPower(controlOutput);
223
224     if (dataLog.size() < 1500) {
225         DataPoint dp;
226         dp.timestamp = micros();
227         dp.sensorValue = sensorValue;
228         dp.selectedSensor = selectedSensor;
229         dp.error = error;
230         dp.controlOutput = controlOutput;
231         dp.strategyUsed = selectedStrategy;
232         dataLog.push_back(dp);
233     }
234   }
235   else {
236     setMotorPower(0.0);
237     if (wasRunning) wasRunning = false;
238   }
239 }
```

web.cpp:

```
1 /*
2 All of web.cpp was created by a Generative AI model and only verified by
  ↪ myself.
3 This is because the web functionality is not the main scope of my
  ↪ project.
4 Therefore, I have offloaded this task to AI.
5 However, all other code, unless stated otherwise, has been programmed by
  ↪ myself
6 */
7
8 /**
9  * @file web.cpp
10  * * Handles all Wi-Fi, mDNS, and Web Server functionality for the
    ↪ ESP32.
11  * * CONFIGURED AS ACCESS POINT (HOTSPOT).
12  */
```

```
13
14 #include <WiFi.h>
15 #include <WebServer.h>
16 #include <ESPmDNS.h>
17 #include <vector>
18
19 // -----------------------------------
20 // Type Definitions (from main file)
21 // -----------------------------------
22
23 struct DataPoint {
24     uint32_t timestamp;
25     float sensorValue;
26     int selectedSensor;
27     float error;
28     float controlOutput;
29     int strategyUsed;
30 };
31
32 // -----------------------------------
33 // External State Variables (defined in main.cpp)
34 // -----------------------------------
35
36 extern volatile bool running;
37 extern volatile int selectedStrategy;
38 extern volatile int selectedSensor; // Now actively used
39 extern float calibMiddle;
40 extern float calibLowVal;
41 extern float calibHighVal;
42 extern std::vector<DataPoint> dataLog;
43 extern unsigned long lastSensorUpdateTime;
44
45 // PID Gains (Externally linked from main.cpp)
46 extern float gain_p;
47 extern float gain_i;
48 extern float gain_d;
49
50 // Helper function from main.cpp
51 String getBufferCSV();
52 void calibrateLowStep(int selectedSensor);
53 void calibrateHighStep(int selectedSensor);
54
55 // -----------------------------------
56 // Web Server Globals
57 // -----------------------------------
58
```

```
59 const char* ssid = "ESP32-Control";
60 const char* password = "12345678";
61
62 WebServer server(80);
63
64 // ------------------------------------
65 // HTML Helper
66 // ------------------------------------
67
68 String getHTML() {
69     String html = R"rawliteral(
70 <!DOCTYPE html>
71 <html>
72 <head>
73     <title>ESP32 Control</title>
74     <meta name="viewport" content="width=device-width, initial-scale=1">
75     <style>
76         body { font-family: Arial, sans-serif; margin: 20px; background:
            ↪  #f4f4f4; }
77         h1 { color: #333; text-align: center; }
78         .container { max-width: 600px; margin: auto; padding: 20px;
            ↪  background: #fff; border-radius: 8px; box-shadow: 0 2px 5px
            ↪  rgba(0,0,0,0.1); }
79         button, select, input { display: block; width: 100%; padding:
            ↪  12px; margin: 10px 0; font-size: 16px; border: none;
            ↪  border-radius: 5px; cursor: pointer; box-sizing: border-box;
            ↪  }
80         input { border: 1px solid #ccc; background: #fff; }
81         select { background: #eee; }
82         button { color: white; }
83         .start { background: #28a745; } .start:hover { background:
            ↪  #218838; }
84         .stop { background: #dc3545; } .stop:hover { background:
            ↪  #c82333; }
85         .calib { background: #ffc107; color: #333; } .calib:hover {
            ↪  background: #e0a800; }
86         .download { background: #17a2b8; } .download:hover { background:
            ↪  #138496; }
87         .btn-blue { background: #007bff; } .btn-blue:hover { background:
            ↪  #0056b3; }
88         .status-box { text-align: center; margin-bottom: 20px; padding:
            ↪  10px; background: #eee; border-radius: 5px;}
89         .status-dot { height: 15px; width: 15px; border-radius: 50%;
            ↪  display: inline-block; margin-right: 8px; vertical-align:
            ↪  middle; }
```

```
        .running { background-color: #28a745; box-shadow: 0 0 8px #28a745; }
        .stopped { background-color: #dc3545; }
        .gains-display { font-size: 0.9em; margin-top: 5px; color: #555; }
        #dataCount { font-weight: bold; color: #555; text-align: center; display: block; margin-top: 10px;}
    </style>
</head>
<body>
    <div class="container">
        <h1>ESP32 Motor Control</h1>

        <div class="status-box">
            Status: <span id="statusDot" class="status-dot stopped"></span> <strong id="statusText">STOPPED</strong>
            <br>Current Strategy: <strong id="stratDisp">)rawliteral";

    html += String(selectedStrategy);

    html += R"rawliteral(</strong>
            <br>Current Sensor: <strong>)rawliteral";

    // Display Sensor Name
    if (selectedSensor == 1) html += "DPS368 (Pressure)";
    else if (selectedSensor == 2) html += "TOF0200C (Distance)";
    else html += "None Selected";

    html += R"rawliteral(</strong>
            <div class="gains-display">
                <strong>Current Gains:</strong>
                P: )rawliteral" + String(gain_p) +
                " | I: " + String(gain_i) +
                " | D: " + String(gain_d) + R"rawliteral(
            </div>
        </div>

        <form action="/start" method="GET" onsubmit="startPolling()"><button class="start" type="submit">Start Experiment</button></form>
        <form action="/stop" method="GET" onsubmit="stopPolling()"><button class="stop" type="submit">Stop Experiment</button></form>

        <span id="dataCount">Recorded Points in Browser: 0</span>
```

```
127
128         <hr>
129         <button class="download" onclick="downloadCSV()">Download
            ↪  Accumulated Data</button>
130
131         <hr>
132         <h3>Configuration</h3>
133
134         <form action="/setSensor" method="GET">
135             <label for="sensor">Select Sensor:</label>
136             <select name="sensor" id="sensor">
137                 <option value="1" )rawliteral";
138                 if(selectedSensor == 1) html += "selected";
139                 html += R"rawliteral(>DPS368 (Pressure)</option>
140                 <option value="2" )rawliteral";
141                 if(selectedSensor == 2) html += "selected";
142                 html += R"rawliteral(>TOF0200C (Distance)</option>
143             </select>
144             <button class="btn-blue" type="submit">Set Sensor</button>
145         </form>
146
147         <form action="/setStrategy" method="GET">
148             <label for="strategy">Control Strategy:</label>
149             <select name="value" id="strategy">
150                 <option value="0">Strategy 0 (Idle)</option>
151                 <option value="1">Strategy 1 (P-Only)</option>
152                 <option value="2">Strategy 2 (ON/OFF)</option>
153                 <option value="3">Strategy 3 (PID)</option>
154             </select>
155             <button class="btn-blue" type="submit">Set Strategy</button>
156         </form>
157
158         <hr>
159         <h3>PID Tuning</h3>
160         <form action="/setPID" method="GET">
161             <label>P Gain: <input type="number" step="0.001" name="p"
                ↪  value=")rawliteral" + String(gain_p) +
                ↪  R"rawliteral("></label>
162             <label>I Gain: <input type="number" step="0.001" name="i"
                ↪  value=")rawliteral" + String(gain_i) +
                ↪  R"rawliteral("></label>
163             <label>D Gain: <input type="number" step="0.001" name="d"
                ↪  value=")rawliteral" + String(gain_d) +
                ↪  R"rawliteral("></label>
164             <button class="btn-blue" type="submit">Update Gains</button>
165         </form>
```

```
166
167         <hr>
168         <h3>Calibration</h3>
169         <p>1. Move rig to LOW position (Last: <strong>)rawliteral" +
            ↪  String(calibLowVal) + R"rawliteral(</strong>):</p>
170         <form action="/calibLow" method="GET"><button class="calib"
            ↪  type="submit">Record Low</button></form>
171         <p>2. Move rig to HIGH position (Last: <strong>)rawliteral" +
            ↪  String(calibHighVal) + R"rawliteral(</strong>):</p>
172         <form action="/calibHigh" method="GET"><button class="calib"
            ↪  type="submit">Record High</button></form>
173         <p><em>Calculated Middle: )rawliteral";
174
175     html += String(calibMiddle);
176
177     html += R"rawliteral(</em></p>
178     </div>
179
180     <script>
181         // -- JAVASCRIPT STREAMING LOGIC --
182         let allData = "timestamp,sensorValue,selectedSensor,
183                   error,controlOutput,strategyUsed\n";
184         let pointCount = 0;
185         let pollInterval = null;
186
187         const isRunning = )rawliteral";
188     html += (running ? "true" : "false");
189     html += R"rawliteral(;
190
191         if(isRunning) {
192             document.getElementById("statusDot").className = "status-dot
                ↪  running";
193             document.getElementById("statusText").innerText = "RUNNING";
194             startPolling();
195         }
196
197         function startPolling() {
198             if(pollInterval) clearInterval(pollInterval);
199             // Fetch chunks every 1 second
200             pollInterval = setInterval(fetchDataChunk, 1000);
201         }
202
203         function stopPolling() {
204             setTimeout(fetchDataChunk, 500); // One last fetch
205             if(pollInterval) clearInterval(pollInterval);
206         }
```

```
207
208         function fetchDataChunk() {
209             fetch('/pollData')
210                 .then(response => response.text())
211                 .then(data => {
212                     if(data.length > 5) { // If valid csv chunk
213                         allData += data;
214                         let lines = data.split("\n").length - 1;
215                         pointCount += lines;
216                         document.getElementById("dataCount").innerText =
                            ↪  "Recorded Points in Browser: " + pointCount;
217                     }
218                 })
219                 .catch(err => console.error("Poll error:", err));
220         }
221
222         function downloadCSV() {
223             const blob = new Blob([allData], { type: 'text/csv' });
224             const url = window.URL.createObjectURL(blob);
225             const a = document.createElement('a');
226             a.setAttribute('hidden', '');
227             a.setAttribute('href', url);
228             a.setAttribute('download', 'experiment_log.csv');
229             document.body.appendChild(a);
230             a.click();
231             document.body.removeChild(a);
232         }
233     </script>
234 </body>
235 </html>
236 )rawliteral";
237     return html;
238 }
239
240 // ------------------------------------
241 // Web Server Route Handlers
242 // ------------------------------------
243
244 void handleRoot() {
245     server.send(200, "text/html", getHTML());
246 }
247
248 void handleStart() {
249     running = true;
250     server.sendHeader("Location", "/");
251     server.send(302, "text/plain", "Starting...");
```

```
252 }
253
254 void handleStop() {
255     running = false;
256     server.sendHeader("Location", "/");
257     server.send(302, "text/plain", "Stopping...");
258 }
259
260 // NEW: Polling Endpoint
261 void handlePollData() {
262     String chunk = getBufferCSV();
263     server.send(200, "text/plain", chunk);
264 }
265
266 void handleCalibLow() {
267     calibrateLowStep(selectedSensor);
268     server.sendHeader("Location", "/");
269     server.send(302, "text/plain", "Low Set");
270 }
271
272 void handleCalibHigh() {
273     calibrateHighStep(selectedSensor);
274     server.sendHeader("Location", "/");
275     server.send(302, "text/plain", "High Set");
276 }
277
278 void handleSetStrategy() {
279     if (server.hasArg("value")) {
280         int strategy = server.arg("value").toInt();
281         selectedStrategy = strategy;
282     }
283     server.sendHeader("Location", "/");
284     server.send(302, "text/plain", "Strategy set.");
285 }
286
287 // NEW: Sensor Selection Endpoint
288 void handleSetSensor() {
289     if (server.hasArg("sensor")) {
290         selectedSensor = server.arg("sensor").toInt();
291     }
292     server.sendHeader("Location", "/");
293     server.send(302, "text/plain", "Sensor set.");
294 }
295
296 // NEW: PID Update Endpoint
297 void handleSetPID() {
```

```
298     if (server.hasArg("p")) gain_p = server.arg("p").toFloat();
299     if (server.hasArg("i")) gain_i = server.arg("i").toFloat();
300     if (server.hasArg("d")) gain_d = server.arg("d").toFloat();
301 
302     server.sendHeader("Location", "/");
303     server.send(302, "text/plain", "PID Updated");
304 }
305 
306 void handleDownload() {
307     server.send(200, "text/plain", "Use the button on the page to
        ↪  download streamed data.");
308 }
309 
310 void handleNotFound() {
311     server.send(404, "text/plain", "404: Not Found");
312 }
313 
314 // -----------------------------------
315 // Public Functions
316 // -----------------------------------
317 
318 void initWebServer() {
319     WiFi.mode(WIFI_AP);
320     Serial.println("Creating Access Point...");
321     WiFi.softAP(ssid, password);
322 
323     IPAddress myIP = WiFi.softAPIP();
324     Serial.print("AP Created! Network Name: ");
325     Serial.println(ssid);
326     Serial.print("Connect to this network and visit: http://");
327     Serial.println(myIP);
328 
329     if (MDNS.begin("esprig")) {
330         Serial.println("mDNS responder started: http://esprig.local");
331     }
332 
333     server.on("/", HTTP_GET, handleRoot);
334     server.on("/start", HTTP_GET, handleStart);
335     server.on("/stop", HTTP_GET, handleStop);
336     server.on("/calibLow", HTTP_GET, handleCalibLow);
337     server.on("/calibHigh", HTTP_GET, handleCalibHigh);
338     server.on("/setStrategy", HTTP_GET, handleSetStrategy);
339     server.on("/setSensor", HTTP_GET, handleSetSensor); // Registered
        ↪  new handler
340     server.on("/setPID", HTTP_GET, handleSetPID);
341     server.on("/pollData", HTTP_GET, handlePollData);
```

```
342     server.on("/download", HTTP_GET, handleDownload);
343     server.onNotFound(handleNotFound);
344 
345     server.begin();
346     Serial.println("HTTP server started");
347 }
348 
349 void handleWebServer() {
350     server.handleClient();
351 }
```

5.4 PYTHON CODE

This section compiles the code used for the Python simulations as well as their graphing.

Simulation code:

```
1  import numpy as np
2  import csv
3  
4  class RotationalOscillator:
5      def __init__(self, angle):
6          self.angle = angle
7          self.angular_velocity = 0.0
8          self.damping = 0.1 # simplified friction
9          self.power = 10.0 # max power multiplier
10 
11     def update(self, control_signal, dt):
12         acceleration = (control_signal * self.power) - (self.damping *
           ↪  self.angular_velocity)
13         self.angular_velocity += acceleration * dt
14         self.angle += self.angular_velocity * dt
15         return self.angle
16 
17 class Sensor:
18     def __init__(self, jitter=0.05, drift=0.0001):
19         self.jitter = jitter # instant noise
20         self.drift = drift # long-term drift
21         self.bias = 0.0 # accumulated drift
22 
23     def read(self, true_angle):
24         self.bias += np.random.normal(0, self.drift)
25         return true_angle + self.bias + np.random.normal(0, self.jitter)
26 
27 class controlStrategy:
```

```
    def __init__(self, kp, ki, kd, dt):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.dt = dt
        self.prev_error = 0
        self.integral = 0

    def compute(self, strategy, setpoint, measured):
        error = setpoint - measured
        p = self.kp * error
        self.integral += error * self.dt
        self.integral = max(min(self.integral, 1.0), -1.0)
        i = self.ki * self.integral
        d = self.kd * (error - self.prev_error) / self.dt
        self.prev_error = error
        # clamp to [-1, 1] always
        if strategy == "PID":
            return max(min(p + i + d, 1.0), -1.0)
        elif strategy == "P":
            return max(min(p, 1.0), -1.0)
        elif strategy == "OnOff":
            if error > 0: return 1.0
            elif error < 0: return -1.0
            else: return 0

# MAIN EXECUTION

dt = 0.001  # 1ms for physics calculations
control_dt = 0.01 # 100 Hz control loop
steps = 600000 # 60 seconds
used_strategy = input("Strategy to simulate (OnOff, P or PID): ") #
  ↪ OnOff, P or PID
kp = 0.3
ki = 0.01
kd = 0.02
control_val = 0
swap_setpoint = True # swap setpoint halfway
setpoint = 0.0

rig = RotationalOscillator(np.radians(45))
sensor = Sensor()
control = controlStrategy(kp, ki, kd, control_dt)

data_rows = []

```

```
print("Running experiment...")

for i in range(steps):
    if swap_setpoint and (i == steps//2):
        setpoint = np.pi/6 # change setpoint to 30°
        print("Setpoint changed")
    if i % (steps // 10) == 0: print(f"Completed {i} steps") # print 10 milestones
    t = i * dt
    measured = sensor.read(rig.angle)
    if (i % (control_dt // dt) == 0): # only compute control every x time
        control_val = control.compute(used_strategy, setpoint, measured)
    true_angle = rig.update(control_val, dt)
    # save data
    data_rows.append([t, true_angle, measured, control_val])

# write data
with open(f"rotational-simulation-{used_strategy}-{int(dt*steps)}s{"-SSwap" if swap_setpoint else ""}.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Time", "True_Angle", "Measured_Angle", "Control_Signal"])
    writer.writerows(data_rows)

print("Done")
```

Graphing code:

```
import matplotlib.pyplot as plt
import csv
import numpy as np
import os

swap_setpoint = True # change depending on what to analyze

user_input = input("Strategy to analyze (OnOff, P or PID): ")
if user_input == "all":
    strategies_to_run = ["OnOff", "P", "PID"]
else:
    strategies_to_run = [user_input]

for strategy in strategies_to_run:
    filename = f"rotational-simulation-{strategy}-60s{"-SSwap" if swap_setpoint else ""}.csv"
```

```
    output_image = f"rotational-simulation-{strategy}{"-SSwap" if swap_setpoint else ""}.png"
    times = []
    true_angles = []
    measured_angles = []

    if not os.path.exists(filename):
        print(f"File not found: {filename}")
        continue

    with open(filename, "r") as f:
        reader = csv.DictReader(f)
        for row in reader:
            times.append(float(row["Time"]))
            true_angles.append(float(row["True_Angle"]))
            measured_angles.append(float(row["Measured_Angle"]))

    # deg for visualization
    true_angles_deg = np.degrees(true_angles)
    measured_angles_deg = np.degrees(measured_angles)

    plt.figure(figsize=(12, 6))
    plt.plot(times, measured_angles_deg, label="Sensor Reading (Noise and Bias)", color="red", alpha=0.3, linewidth=0.5)
    plt.plot(times, true_angles_deg, label="True Orientation", color="#007acc", linewidth=2)
    if swap_setpoint:
        plt.hlines(0, min(times), 30, color='gray', linestyle='--', linewidth=1.5, label="Target (0°, 30°)")
        plt.hlines(30, 30, max(times), color='gray', linestyle='--', linewidth=1.5)
        plt.axvline(30, color='green', linestyle=':', linewidth=1.5, label="Setpoint Change")
    else:
        plt.axhline(0, color='gray', linestyle='--', linewidth=1.5, label="Target (0°)")
    # proper naming
    used_strategy = ""
    if strategy == "OnOff": used_strategy = "On-Off"
    elif strategy == "P": used_strategy = "Proportional"
    elif strategy == "PID": used_strategy = "PID"

    plt.title(f"Rotational Oscillator Response ({used_strategy} Control{", Swapping Setpoint" if swap_setpoint else ""})", fontsize=14)
    plt.xlabel("Time (seconds)", fontsize=12)
```

```
    plt.ylabel("Angle (Degrees)", fontsize=12)
    plt.legend(loc="upper right")
    plt.grid(True, which='major', linestyle='-', alpha=0.6)
    plt.grid(True, which='minor', linestyle=':', alpha=0.3)
    plt.minorticks_on()
    plt.ylim(min(true_angles_deg) - 10, max(true_angles_deg) + 10)
    plt.tight_layout()
    plt.savefig(output_image, dpi=600)
    print(f"Done processing {strategy}")
```

5.5 DATA EXTRACTED FROM SIMULATION

The CSVs containing the data for the simulations have not been included, as the total amount of lines from all of them would nearly reach one million. However, all the data is available through the QR code at the start of the research paper, in the project's GitHub repository.

5.6 WEB FUNCTIONALITY PROMPT

The following prompt was used with Gemini's 3 Pro (Thinking) model:

> Act as an embedded C++ expert and generate a source file named web.cpp for an ESP32 project. The ESP32 must operate as a WiFi Access Point with the SSID "ESP32-Control" and use the WebServer.h library.
> The code must rely on extern variables to interface with the main application, specifically: volatile booleans for running state, integers for selectedStrategy and selectedSensor, floats for calibration values (low, middle, high), and floats for PID gains (P, I, D).
> Create a function getHTML() that returns a responsive HTML dashboard in a raw string. The dashboard should display the current system status, the specific sensor name, and the current PID values. Include HTML forms to allow the user to change the sensor, select a control strategy (0-3), and update PID gains.
> Implement a client-side data logger using JavaScript: when the experiment is running, the browser should poll the endpoint "/pollData" every second to retrieve CSV data chunks (generated by an external helper function getBufferCSV), accumulate them in a JavaScript variable, and provide a button to download the full history as "experiment_log.csv".
> Implement all necessary server request handlers to update the extern variables and route the CSV data.