

Article

Toward Autonomous Ecological Monitoring with Betaflight FPV Drones: Test-Stand Validation of a Lightweight Visual Servoing Pipeline

Mykola Dyvak ¹, Kostiantyn Verhun ¹, Andriy Melnyk ^{1,2,*} and Piotr Pikiewicz ^{3,*}

¹ Department of Computer Science, West Ukrainian National University, 11 Lvivs'ka Str., 46000 Ternopil, Ukraine; mdy@wunu.edu.ua (M.D.); k.verhun@gmail.com (K.V.)

² Department of Clinical Engineering, Academy of Silesia, 40-555 Katowice, Poland

³ Department of Mathematical Methods in Technics and Informatics, Silesian University of Technology, 44-100 Gliwice, Poland

* Correspondence: ame@wunu.edu.ua (A.M.); piotr.pikiewicz@polsl.pl (P.P.)

Abstract

Small first-person-view (FPV) drones provide a flexible platform for ecological and environmental monitoring tasks, including the localization of pollution sources, inspection of contaminated areas, and low-disturbance observation of wildlife. However, Betaflight, one of the most widely used flight-control platforms for FPV drones, does not provide native support for autonomous vision-based target tracking. This study presents a lightweight onboard visual servoing framework that integrates real-time target tracking, image-based feedback control, and direct interaction with a Betaflight flight controller. Target tracking is performed using NanoTrack on a Raspberry Pi 5, while the target position estimated from the video stream is converted into a visual tracking error and processed by a PI controller to generate yaw-control commands. These commands are transmitted to the flight controller through the MSP Override mechanism, enabling closed-loop vision-based target centering without modifying the Betaflight firmware. A unified FlightController abstraction is introduced to support both simulation and physical deployment using the same high-level control architecture. The simulation environment combines ArduPilot SITL, Gazebo, and MAVLink, whereas the experimental platform employs Betaflight and the MSP protocol. The proposed framework is evaluated through a progressive validation procedure ranging from software-in-the-loop simulation to experiments with a physical FPV drone mounted on a yaw-only test stand. Experimental validation consists of two data-logged test-stand runs totaling 111 s, during which the system tracked a hand-moved visual target and commanded the drone to rotate in real time to keep it centered. The primary focus of this work is a proof-of-concept demonstration of end-to-end system integration, with the main contribution lying in the integration of existing perception, control, and communication components into a functional vision-to-command pipeline for this class of hardware. Closed-loop operation is demonstrated indoors using high-contrast visual targets and yaw-axis rotation. Therefore, the presented results should be interpreted as validation of the system architecture and integration concept rather than as a comprehensive evaluation of autonomous target tracking under field conditions. Quantitative evaluation using repeatable target trajectories, extension to free-flight operation, and field trials involving ecologically relevant targets constitute the next stage of this research. The proposed framework is intended as a computationally light-

Academic Editor: Yong Liu

Received: 8 September 2026

Revised: 11 October 2026

Accepted: 13 October 2026

Published: 16 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

weight and hardware-compatible foundation for further development of vision-based target tracking and visual servoing on Betaflight-controlled FPV drones for ecological and environmental monitoring applications.

Keywords: visual servoing; target tracking; object tracking; FPV drone; Betaflight; onboard computing; NanoTrack; image-based control; closed-loop control; ecological monitoring

1. Introduction

Small First-Person-View (FPV) drones have become a widely accessible platform for environmental monitoring, ecological field work, and safety applications. The cheap and accessible (compared to industrial UAVs) hardware, light weight, and relatively simple mechanical structure make them attractive for tasks where a larger industrial UAV would be too heavy, too expensive, or too disruptive to the environment. Environmental and ecological monitoring is one of the primary drivers of such developments [1]: drones have been used to identify and map pollution sources and track their spread across water bodies and terrain, monitor contaminated or disaster-affected sites in areas inaccessible on foot, assess the ecological state of remote ecosystems, and detect industrial discharge or illegal dumping from the air. Wildlife monitoring is another well-established application, from counting animals in difficult terrain [2] to observing species behavior at a low-disturbance distance [3]. All of these tasks share a common pattern: the drone is expected not only to fly, but also to react to what it sees—follow a target object, keep a point of interest in view, or approach an anomaly for closer inspection. Having on-board autonomy in addition to the base flight stack makes the process less reliant on continuous human involvement and leads to a more scalable and reliable monitoring workflow.

At the same time, the dominant firmware on FPV drones is Betaflight [4], which was originally developed for manual acrobatic flight. Betaflight provides sufficient low-level stabilization but no built-in autonomy API. Autonomous behavior on a Betaflight drone therefore has to be implemented by a companion computer that interacts directly with the flight controller at the RC-channel level. Betaflight exposes a mechanism called MSP Override [5] that allows selected radio channels to be replaced with values coming from a serial port instead of from the RC receiver. This is the mechanism we use for switching control between radio remote control and programmatical closed-loop control system running on Raspberry Pi computer.

Another practical issue is testing. Testing an autonomous control solution for a UAV requires a proper environment and is tied to the risk of losing or damaging the drone. For this reason it is important to be able to test the developed system in a simulated environment first, so that a large part of the problems can be identified and fixed before any real flight. However, most academic simulation tools (ArduPilot SITL [6], PX4 SITL, Gazebo-based stacks [7]) target MAVLink-based autopilots, while Betaflight uses a completely different protocol [8,9]. As a result, code written against a simulator normally cannot be reused on a real FPV drone without a rewrite.

Taking both points into account, we present a visual servoing pipeline for Betaflight-based FPV drones built around a flexible architecture that keeps the control logic independent of the underlying flight stack, so the same code runs against both in a simulated environment and on a real Betaflight drone without modification. The pipeline is first validated in simulation, then deployed on a real drone fixed to a yaw-only test stand, where closed-loop visual centering is demonstrated end to end (three observed runs and two

data-logged runs, Section 6.2). Target tracking uses NanoTrack [10], which was evaluated and selected for this hardware in prior work [11].

The contributions of this work are at the level of system integration:

- A FlightController abstraction that decouples the control logic from the underlying flight stack, so that identical control code runs both against a MAVLink-based simulator and on a real Betaflight drone over MSP, without modification.
- An end-to-end integration of an existing lightweight tracker, a standard PI regulator and Betaflight's MSP Override into a closed-loop visual servoing pipeline running on a CPU-only companion computer—a platform class that the visual-servoing literature, which targets ArduPilot/PX4, largely does not address.
- A staged simulation-to-hardware integration procedure that reuses the same binary at every stage, and a proof-of-concept demonstration on a yaw-only test stand on which the real drone physically centers on a hand-moved target in closed loop.

The object tracker is employed as an established component, while the control strategy is based on a standard PI regulator with conventional anti-windup and rate-limiting mechanisms. The main contribution lies in integrating these components into a complete end-to-end framework that enables low-cost FPV hardware to close the visual feedback loop in real time.

2. Related Work

2.1. UAVs for Ecological and Environmental Monitoring

Unmanned aerial vehicles have become an established tool for ecological and environmental monitoring. They have four practical advantages over the traditional alternatives, manned aircraft and satellite imagery. They fly low, so the ground resolution of the imagery is much higher than that of satellite data. A flight can be repeated whenever the observation requires it, instead of waiting for a satellite pass or arranging an aircraft. Sites that are hazardous, remote, or impractical to reach on foot can be surveyed without sending a person there. Finally, the cost per survey is considerably lower [1,12].

The adoption of UAVs in ecology is recent, and its main stages can be traced in the literature. The earliest peer-reviewed assessments of small UAVs for wildlife research appeared in the mid-2000s [13,14]. Low-cost autonomous airframes brought the method within reach of field biologists around 2012, and the term drone ecology dates from the same period [15]. By 2013 lightweight platforms were being described as an imminent change in spatial ecology [16], and a ten-year retrospective by the same authors considers that prediction fulfilled and the technology established in ecological practice [17]. The open problems have accordingly moved from the airframe itself to what the drone does with what it sees.

Present-day applications cover a wide range of domains. Vegetation and forest health assessment is now routine [18], as is crop disease detection from multispectral imagery [19]. Water-related applications include inland water quality assessment [20], coastal erosion and risk monitoring [21], and detection of floating litter [22]. Emission sources are surveyed directly, for example methane flux above landfill sites [23]. Drones are also used in disaster response [24], forest fire surveillance [25], and humanitarian demining [26]. In wildlife research they support population counts that are more accurate than ground-based observation [2], as well as species identification, habitat assessment, and behavioral observation [27].

Most of this work follows the same mission pattern: a pre-planned trajectory, usually a lawnmower pattern, is flown over a scene that does not change during the flight, and the imagery is processed after landing [18]. A growing group of targets does not fit this pattern. An animal or a herd, drifting debris, a spreading plume, or an advancing fire front

is mobile or short-lived, and the task is then not to cover an area but to stay on the object. Detecting such targets in aerial video is largely solved: deep detection and tracking models are mature and widely used [28,29]. They are, however, applied almost exclusively offline, to recordings already brought to the ground. A recent review identifies the remaining step as making drones “act autonomously, adjusting flight paths to watch or follow individuals or groups” [17], and the first open and affordable platforms built for autonomous behavioral filming are only now appearing [30].

Following a moving target manually is not a good substitute. Manual piloting limits the duration and the consistency of an observation, and it tends to bring the drone closer to the target than the task requires. This is an ecological problem and not only an operational one: approach distance, altitude, speed, and acoustic signature are the main factors determining behavioral and physiological disturbance in the observed animals [3,31], and minimum separation distances can now be derived per species instead of by rule of thumb [32]. Since the required separation is now a specific distance rather than a rough guideline, maintaining it throughout a flight becomes a control task rather than a matter of pilot judgment.

The second constraint is computational. Field sites often have no reliable data link, and in disaster or remote-terrain scenarios ground infrastructure cannot be assumed at all [24], so the perception that steers the drone has to run on board. On-board inference has been demonstrated in this setting: a GPU-accelerated companion computer performing detection and tracking in flight [33], and a wildfire detector running on a Jetson-class board carried by the UAV [25]. Systems that offload processing to an edge server instead depend on network coverage that a field deployment may not have [22]. In all these cases the limiting factor is the compute budget of a small airframe rather than the algorithm itself.

What the existing autonomous platforms have in common is the hardware they are built on: research-grade autopilots exposing MAVLink, together with GPU-class companion computers. Surveys of practitioners who fly drones in the field report a strong interest in computer-vision automation, and name cost and complexity as the main barriers to adoption [34]; affordability is an explicit design goal of the newer ecological platforms [30]. The cheapest and most widely available airframes, however, are FPV quadcopters running Betaflight, for which documented autonomy paths are few and rely on GPU-class companion boards, as in autonomous drone racing. The present work takes a first, integration-level step towards closing this gap: a lightweight Siamese tracker and a PI controller acting on the horizontal image error, with MSP Override as the control channel on a CPU-only companion computer.

2.2. Visual Servoing and Closed-Loop Control for UAVs

Multirotor control is almost always organized as a cascade of two loops [35–37]. The inner loop stabilizes attitude or angular rate; it runs on the flight controller at a high rate, needs little information about the airframe, and is what makes the vehicle flyable at all. The outer loop commands position or velocity, runs slower, and requires a state estimate. The same division appears in the flight stacks used in practice, but not to the same depth. ArduPilot and PX4 implement both loops, so a companion computer commands them at the position or velocity level. Betaflight implements only the inner one [4]: it stabilizes attitude and rate and leaves everything above that to the pilot. This is what makes an added outer loop both possible and necessary here, and a companion computer can supply it without touching the stabilization running underneath.

At the inner loop, classical linear control is still the operational default. A recent survey synthesizes about 290 works on PID control of quadrotors [38], and PID-family regu-

lators are what production flight stacks ship. The reasons are practical rather than theoretical: no dynamic model has to be identified, the computation is negligible, and the gains can be inspected and retuned in the field. The same holds when the error comes from an image instead of a position estimate, where a PID-like regulator maps naturally onto attitude commands [39].

Model-based methods improve on this, at a cost. A comparative study of five attitude-stabilization schemes on one platform—PID, LQR, MPC, feedback linearization, and sliding-mode control—ranks sliding-mode control highest on response speed and robustness [40]. Hybrid backstepping and sliding-mode designs address the chattering that sliding-mode control suffers from, and have been validated under payload variation and wind [41]. Nonlinear model predictive control performs comparably to differential-flatness-based control in agile flight, although the benefit of the online optimization depends on the task [42]. All of these methods require an identified dynamic model of the airframe, and model predictive control additionally requires enough computing power to solve an optimization problem at every control step. Neither is available on the platform targeted here: a commodity FPV quadcopter is assembled from off-the-shelf parts with no published or measured dynamic model, and the small companion computer it can carry is not sized for online optimization.

Learning-based control occupies the other end of the range. A reinforcement-learning policy has beaten human world champions in FPV drone racing, on the same class of airframe used in this work [43]. A survey of learning-based multirotor algorithms notes that only a part of this work is realistically implementable in real time on board [44]. The cost is simulation infrastructure, domain-randomization pipelines, and GPU-class training, and what is given up is the interpretability and the in-field retunability of a regulator with two gains.

Visual servoing is a separate paradigm within this landscape. Image-based visual servoing (IBVS) controls a robot by minimizing an error defined directly in the image plane [45,46], in contrast to position-based visual servoing, which first reconstructs a pose and then controls it. The practical consequence is that IBVS does not require global localization, which is the property our system depends on. Recent surveys extend the classical taxonomy to learning-based servoing [47] and compare the convergence behavior of the architectures in image space and in task space [48].

On UAVs specifically, vision-based navigation is an active field, covering localization, mapping, and obstacle avoidance in GPS-denied conditions [49]. Deep reinforcement learning has been applied to keeping a target inside the field of view by adapting the velocity gains online [50], which is the same failure mode we handle with a deadband and an angular-rate limit. Complete detect-track-follow systems have been demonstrated. One combines YOLOv8 detection with BoT-SORT and ByteTrack multi-object tracking, and closes the following loop with proportional controllers on yaw rate and altitude together with a PI controller on horizontal velocity [51]. Another uses colour segmentation for detection, an extended Kalman filter for state estimation, and a PID controller for following [52]. Both reach the same architecture we use—lightweight perception driving a classical regulator—but on much larger hardware: the first runs on a Pixhawk flight controller with an NVIDIA Jetson Xavier NX and an RGB-D camera, carried by a hexacopter frame [51]. A MAVLink-class autopilot and a GPU companion board are the norm in this literature.

Our system sits at the low-complexity end of this range, and this is a deliberate choice rather than an omission. The companion computer is a CPU-only Raspberry Pi 5 running the tracker at about 30 FPS, telemetry from the flight controller arrives at about 6 Hz over USB CDC, and Betaflight, although it maintains an internal attitude estimate for its own stabilization (used, for example, in ANGLE mode), provides no position or velocity estimate and no high-level autonomy interface comparable to that of ArduPilot or PX4. Under

these constraints the model-based and the learning-based options have nothing to build on: there is no identified model, no pose, and no GPU. A PI regulator on the image error, with a deadband and a rate limit, is a practical choice for this hardware. At present the loop is closed on the horizontal image error alone and acts on yaw, which is what the yaw-only test stand permits; applying the same regulator to the vertical error, and through it to pitch and throttle, is the immediate next step. A sliding-mode or predictive controller would settle faster and reject disturbances better than a PI regulator [40], but that margin is only available to a system that has the model and the computing power needed to obtain it. A PID-family regulator is in any case what production flight stacks use [38], and the two vision-based following systems described above reach the same conclusion independently [51,52]. The contribution of this work is accordingly one of system integration and engineering rather than of control theory (Section 1): a working chain of tracker, controller and command path that makes a Betaflight drone turn towards what its camera sees, integrated and demonstrated end to end from simulation to a real airframe on a test stand.

2.3. Companion Computer Autonomy on FPV Drones

ArduPilot and PX4 expose MAVLink [53] for high-level companion control; Betaflight uses MSP, a minimal protocol operating at the level of raw RC channel values where all higher-level behavior must be implemented on the companion side. The autonomy mechanism on Betaflight is MSP Override: the companion sends MSP_SET_RAW_RC frames with a channel mask to replace selected RC channels [4,5], an approach also used for autonomous drone racing [54]. Modular frameworks such as Agilicious [8] and aerial-autonomy-stack [9] target research-grade platforms: aerial-autonomy-stack assumes MAVLink throughout, and while Agilicious does provide a serial bridge to Betaflight flight controllers, neither exposes a single interface behind which the same control code runs unchanged against a MAVLink-based simulator and a Betaflight airframe. That equivalence is what the abstraction presented here provides.

2.4. Lightweight Visual Tracking on Edge Devices

The selection of a suitable tracker for embedded deployment was addressed in our prior work [11], which evaluated deep trackers on a Raspberry Pi 5 and identified NanoTrack [10] as the best balance of accuracy and inference speed for this hardware class. NanoTrack uses a MobileNetV3-based Siamese backbone of about 1.1 MB and is available through OpenCV as cv::TrackerNano [55], which is the integration point we use in this paper.

2.5. Simulation Environments

Gazebo [56] is the most widely used simulator for UAV research, usually combined with a flight-stack-specific bridge. RotorS [7] popularized the modular MAV simulator pattern; ArduPilot SITL [6] provides software-in-the-loop execution of the real autopilot code with a Gazebo backend for visuals and dynamics. We reuse this stack as-is for the simulation side of our system and communicate with the simulated vehicle through MAVSDK [53]. No equivalent SITL exists for Betaflight, which is one of the reasons the abstraction layer is needed.

3. System Architecture

3.1. Hardware Platform

The physical setup is a small FPV quadcopter built around a SpeedyBee F405 V3 (SpeedyBee, Shenzhen, China) flight controller (Betaflight firmware, BMI270 gyro, built-

in barometer) and a Raspberry Pi 5 companion computer (Raspberry Pi Ltd., Cambridge, UK) [57]. For these experiments the Raspberry Pi is connected to the flight controller over USB, which provides a convenient single-cable setup; a hardware UART connection is the planned next step, as it offers substantially lower round-trip latency and is more reliable in a vibrating airframe. The video feed is provided by an analog camera with an analog-to-digital converter connected to a USB port on the companion computer. An ELRS radio link (Radiomaster TX12 MKII (RadioMaster, Shenzhen, China)) is kept on the drone at all times, so that the human pilot retains override at any moment—this is important for safety, and it is also the way the drone is armed. The RC transmitter is used to switch MSP Override on and off via an AUX channel.

3.2. FlightController Abstraction Layer

The central design choice of the system is a small abstract C++ interface, *FlightController*, that hides the flight stack details from the control logic. Three backends implement this interface (Figure 1):

- *MAVLinkFC*—wraps MAVSDK and is used to talk to ArduPilot SITL running in Gazebo. Used for simulation only. Optional build dependency (WITH_MAVLINK).
- *BetaflightFC*—talks MSP over a POSIX serial port to a real Betaflight flight controller. Used on real hardware.
- *PrintCommandsFC*—a debug backend that prints commands to stdout in one of several formats (raw, msp, mavlink). Used for dry runs without any hardware.

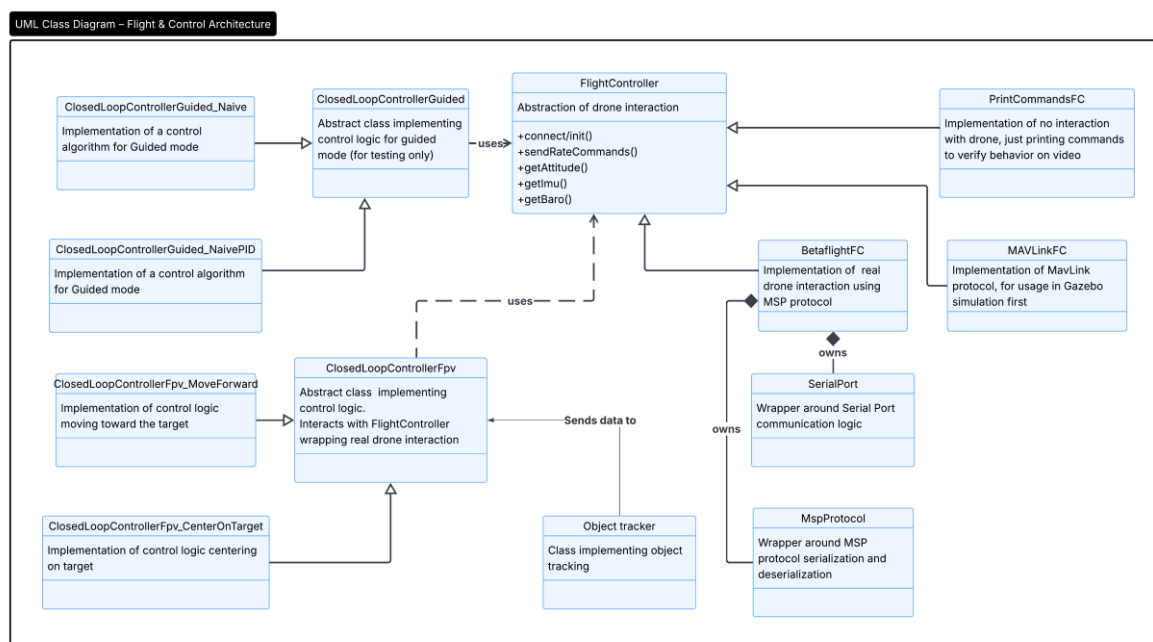


Figure 1. Class diagram of the flight and control architecture. The *FlightController* interface has three backends: *MAVLinkFC* (simulation), *BetaflightFC* (real hardware via MSP), and *PrintCommandsFC* (dry-run).

The *FlightController* abstraction provides two-way communication with a flight controller: sending control commands and receiving sensors data.

All control algorithms work against the *FlightController* interface only; they do not depend on *MAVLink* or *MSP* implementations directly. Because of this separation, transitioning a controller from simulation to real hardware requires no modifications to the control code itself.

3.3. MSP Override Configuration

Betaflight supports a feature called MSP Override which allows the companion computer to replace selected RC channels with values coming from the serial link. Configuration requires two main steps. First, the override mask is set via the Betaflight CLI: `set msp_override_channels_mask = 15`, where mask 15 (0b00001111) covers exactly Roll, Pitch, Yaw, and Throttle (bits 0–3). Second, MSP Override itself is bound to an RC switch, so that the pilot can enable and disable companion-computer control from the radio. ARM and the main flight mode switch are deliberately not included in the mask—the human pilot is always the one who arms the drone and can always take back control by flipping the override switch.

3.4. Software Pipeline

The high-level data flow is shown in Figure 2. A dedicated command thread sends cached RC values to the flight controller independently of the camera frame rate. This is important because the tracker latency is variable (and sometimes high on low-contrast frames), and Betaflight will trigger its MSP failsafe if it does not receive an `MSP_SET_RAW_RC` frame often enough. Decoupling the command thread from the vision thread removes this source of failure. A second thread polls sensor telemetry from the flight controller.

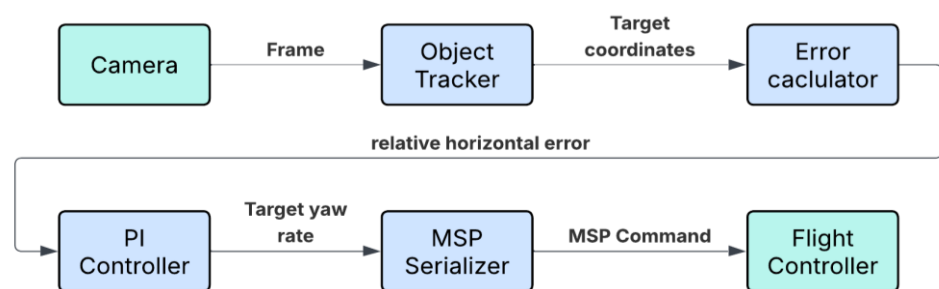


Figure 2. Data flow on the Raspberry Pi. Each camera frame is processed by object tracker to produce a bounding box; the horizontal error is fed to the PI controller, which outputs a yaw rate command sent to the flight controller via MSP Override at 50 Hz.

The pilot arms the drone and engages MSP Override via an RC switch, at which point the companion computer takes over the yaw channel in closed loop; flipping the switch back instantly returns full control to the pilot.

4. Tracking and Control

4.1. NanoTrack Integration

NanoTrack is integrated through the OpenCV `cv::TrackerNano` API [55]. The model is a MobileNetV3-based Siamese network exported to ONNX, about 1.1 MB in total; on a Raspberry Pi 5 it runs at approximately 30 FPS at the tracker’s native 640×480 input resolution [11]. The tracker is initialized from a bounding box selected interactively on the first frame, or passed via a command-line flag for repeatable experiments.

The output of the tracker for each frame is a bounding box on the tracked target. We reduce it to a single 2D target center and normalize the horizontal coordinate to $[-1,1]$ (left edge to right edge of the image).

4.2. CenterOnTarget PI Controller

Let e_x be the normalized horizontal error between the target center and the image center:

$$e_x = \frac{2 \cdot (x_{\text{target}} - x_{\text{image center}})}{W_{\text{image}}} \quad (1)$$

where W_{image} is the image width. A discrete PI control law is applied at each frame step k :

$$\omega_{\text{yaw}}[k] = K_P e_x[k] + K_I \Delta t \sum_{j=0}^k e_x[j] \quad (2)$$

with an anti-windup clamp on the accumulator: after each update the integral state is constrained to $[-C_I, +C_I]$ with $C_I = 0.3$ s, limiting the integral's maximum contribution to $\pm K_I C_I = \pm 0.03$ rad/s. Δt is the time elapsed since the previous controller update, measured at each iteration with a monotonic clock, so the integral term remains correctly scaled when the update interval varies with tracker latency. When the target is inside the deadband ε , the output is intended to be zeroed and the integrator reset to zero immediately. The controller as implemented, and as used in all experiments reported here, evaluates this centered condition on both image axes simultaneously: the output is zeroed and the integrator reset only when the target is inside the deadband both horizontally and vertically. In the yaw-only configuration the vertical error is not controlled, and on the test stand it was rarely inside the deadband, so the zero-and-reset branch was effectively inactive during the hardware runs. While the horizontal error is inside the deadband, the yaw command therefore does not drop to zero; it remains bounded by the proportional contribution at the deadband edge plus the clamped integral contribution. The output ω_{yaw} is clipped to $[-\omega_{\text{max}}, +\omega_{\text{max}}]$ and mapped to the Betaflight PWM range:

$$\text{PWM}_{\text{yaw}} = 1500 + \frac{500}{\omega_{\text{fs}}} \cdot \omega_{\text{yaw}} \quad [\mu\text{s}] \quad (3)$$

where $\omega_{\text{fs}} = 0.5934$ rad/s ($\approx 34^\circ/\text{s}$) is a fixed scaling constant of the companion software, chosen in practice to keep the rotational speed lower; Equation (3) is therefore a command encoding rather than a model of the flight controller. The yaw rate that Betaflight derives from the received stick value is determined by its rate profile and was not measured experimentally. The Betaflight version used is 2025.12.2, the rate profile type is "Actual". A rate-of-change limiter of 0.1 rad/s² additionally constrains the per-step change in ω_{yaw} , smoothing abrupt command steps and reducing mechanical shock during stand tests. The resulting value is clamped to $[1000, 2000]$ and placed into the MSP_SET_RAW_RC payload. The remaining RC channels (roll, pitch, throttle) are held at fixed neutral values during closed-loop operation; throttle is set to 0.50 of full scale, an empirically chosen value that sustains approximate hover on the test stand.

The integral term is included so that a small, persistent error is eventually eliminated: a pure P controller stops increasing the yaw command once ω_{yaw} is balanced by a disturbance opposing the rotation, such as friction in the test stand, leaving the target visibly off-center; the integral accumulates that residual error and pushes the drone the rest of the way. In the current version, a derivative term is omitted because differentiating tracker output amplifies the discrete bounding-box jumps that NanoTrack occasionally produces; it will be added later if further experiments reveal the need.

The specific values used for the experiments reported in this paper are summarized in the Table 1.

Table 1. PI controller parameters and loop rates used for the two data-logged test-stand runs.

Parameter	Value
K_P (proportional gain)	0.4 rad/s
K_I (integral gain)	0.1 rad/s ²
Integral clamp	± 0.3 s
ε (deadband)	0.02

$\omega_{\max}$ (rate limit)	0.1 rad/s
Rate-of-change limit	0.1 rad/s ²
Hover thrust	0.5
Camera resolution	640 × 480
Camera loop rate	30 FPS
Command thread rate	50 Hz

Betaflight is run in ANGLE mode; the firmware's internal flight controller logic stabilizes roll and pitch automatically, so the companion computer only needs to drive the outer yaw loop.

4.3. Dual-Thread Architecture

The BetaflightFC backend uses two background threads: a command thread that sends cached RC values at 50 Hz independently of the camera loop, and a telemetry thread that polls sensor data at ~6 Hz. Decoupling the command thread from the camera loop prevents MSP failsafe during high-latency tracker frames.

5. Simulation Environment

The simulation side of the system is built on ArduPilot SITL [6] with a Gazebo [56] world; the overall component layout is shown in Figure 3. SITL runs the real ArduPilot code as a desktop process and simulates the sensors and the vehicle dynamics. Communication with the simulated vehicle uses MAVLink over UDP via MAVSDK [53]; the camera feed from Gazebo is delivered over a GStreamer pipeline (UDP port 5600) to the same C++ executable that is used on the real hardware. SITL runs a standard ArduCopter quadrotor frame, and the simulated camera is configured to match the 640 × 480 resolution used on the real platform, so that the image-error normalization and PI gains carry over unchanged between the two environments.

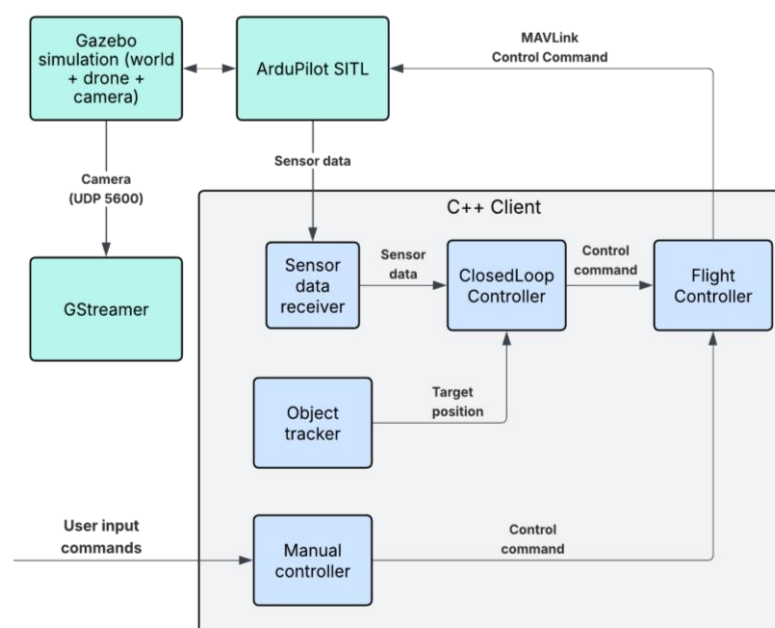


Figure 3. Simulation stack components. Gazebo provides world, drone model, and camera; ArduPilot SITL runs the real autopilot code and exchanges MAVLink control commands and sensor data with the C++ client; GStreamer delivers the camera stream to the object tracker inside the client.

The CenterOnTarget PI controller described in Section 4.2. was validated in this environment against both static and dynamic targets before being deployed on real hardware. The transition from simulation to real hardware did not require any changes to the control code.

The simulation reproduces the geometry of the tracking problem and the full software path, but not the physical conditions of the airframe, and the differences matter when the two sets of results are compared. Four effects are absent by construction: the latency of the USB CDC link between the companion computer and the flight controller, which in simulation is replaced by a UDP socket on the same host; mechanical vibration transmitted from the propellers into the camera mount; the degradation introduced by the analog camera and its analog-to-digital converter, since Gazebo delivers a clean synthetic image; and the idealized yaw dynamics of the simulated airframe, which has neither prop-wash, motor thrust asymmetry, nor the stiction of a physical test stand. Together these account for the visibly higher noise level in the hardware traces of Section 6.2. relative to their simulated counterparts.

6. Experimental Validation

6.1. Progressive Testing Methodology

Validation was performed step-by-step with gradually increased complexity, so that each stage would fail in a predictable way and surface only a small number of issues at once.

The testing progressed through four main stages:

1. Simulation. The control algorithm was first validated in the simulated environment using an artificial scene generated in Gazebo, with the drone running on ArduPilot SITL and MAVLink for communication.
2. Dry run on real hardware. The PrintCommandsFC backend was used to run the full pipeline on the Raspberry Pi with a real USB webcam but no flight controller, to check that camera, tracker, and controller work together end-to-end on the target device.
3. Bench test with flight controller, propellers off. The Raspberry Pi was connected to the SpeedyBee F405 V3 over USB. Simple test scenarios were used to control individual motors via MSP and verify the protocol implementation. Then MSP Override was enabled and the commands coming out of the closed loop were checked on the FC side without actually producing thrust.
4. Test stand with propellers on. The drone was placed on a test stand that allowed it to rotate freely about its yaw axis, with pitch and roll restricted to small ranges and translation prevented, MSP Override was engaged, and the real closed-loop tracking was exercised with a LiPo battery and a 100% motor output limit.

For each stage the same binary and the same control code were used; only command-line flags changed.

The remainder of this subsection describes each stage in more detail, including the integration problems it exposed, since these are the substance of what the escalation was designed to find.

Stage 1: simulation. Runs were performed in Gazebo with the vehicle in ACRO mode, using two scenario families: a sequence of static targets selected in turn, and a single continuously moving target. Sixteen runs with valid closed-loop data were recorded across the simulation campaign.

Stage 2: dry run. The dry run was executed twice, first on a desktop machine and then on the Raspberry Pi 5, with an identical command line, an identical USB webcam and the same target, so that the embedded result could be compared against a known-good baseline rather than interpreted on its own. Both produced the same behavior: the tracker

followed the target and control commands were generated at the expected rate, with no hardware errors or crashes.

Stage 3: bench test with the flight controller. This stage exposed the integration defects that are invisible in simulation. The first concerned mode consistency: the control mode selected in the application must correspond to the mode in which the vehicle is actually operating, since the simulation and hardware backends expose different command interfaces and only the matching one produces traffic on the link. With the modes in sync, MSP Override was verified end-to-end: the transmitted yaw channel moved in the expected direction for a target displaced to the left, and the motor note changed audibly when the override switch was engaged, confirming that the flight controller was acting on the injected channels. Two further observations were recorded at this stage: the software disarm channel could be left inactive without consequence, since arming is managed from the transmitter and the ARM channel is deliberately excluded from the override mask; and the integral term saturated against its clamp within a few seconds on a stationary bench target, which is the expected behavior for a persistent error and motivated the anti-windup treatment described in Section 4.

Stage 4: test stand. The final propellers-on stage exposed a few more failures. With the motor output limited to 15 % in the flight controller configuration, thrust was insufficient to produce any yaw response. Raising the limit to 100 % restored yaw authority, but with the vehicle powered from a bench supply the airframe still did not rotate: the fixed throttle of 0.50 of full scale yields only 45–47 % output after the neutral-channel mapping, which does not generate a usable torque differential against the constraint of the stand. Substituting a flight battery for the bench supply, with the flight controller in ANGLE mode, produced the first successful closed-loop runs, and all reported stand results were obtained in that configuration. The stand itself is shown in Figure 4.

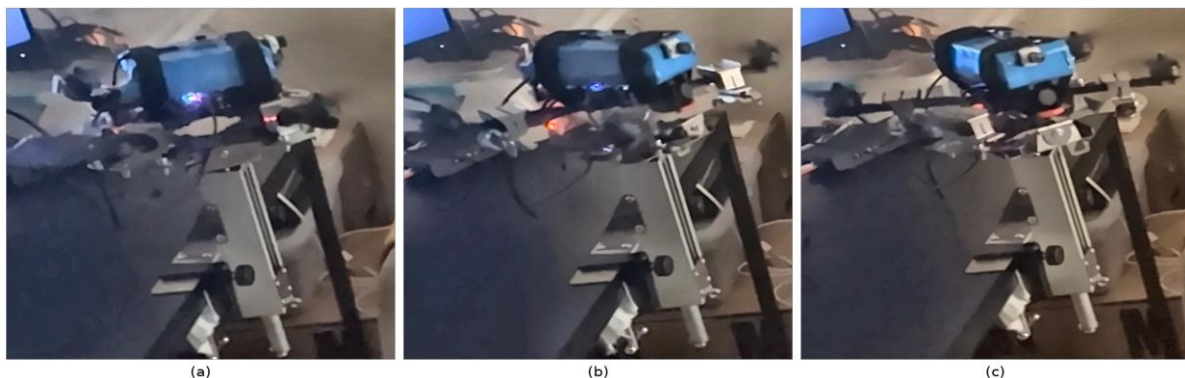


Figure 4. The drone mounted on the yaw-only test stand, at three successive yaw orientations during closed-loop tracking of a hand-moved target. (a) Initial yaw orientation, with the camera; (b) intermediate orientation during the rotation; (c) final yaw orientation. The airframe is clamped to a vertical post that permits unrestricted rotation about the yaw axis, allows only small pitch and roll deflections (within 12° and 15° , respectively) and prevents translation.

Test stand. All propellers-on runs were performed on a commercially available drone test stand, designed for 7–15-inch airframes and made of stainless steel and aluminum. The stand is fixed to the table top with two metal clamps; it has a mass of 1.8 kg and overall dimensions of $250 \times 400 \times 120$ mm. It permits unrestricted (360°) rotation about the yaw axis, limits pitch and roll to ranges of 12° and 15° , respectively, and prevents translation. The airframe is mounted on a moving platform with a mass of 1.25 kg, which rotates with it and therefore adds to the yaw inertia of the system.

Behavior on loss of track. When the tracker reports no valid target, the current implementation returns a zero command: the commanded yaw rate is set to zero. The integral accumulator is neither cleared nor decayed on this path, so it is frozen at its last value and resumes from there if the target is reacquired. The frozen integral contribution is bounded by the anti-windup clamp to ± 0.03 rad/s, and the pilot can take back full control at any time with the override switch. Refining this behavior—in particular deciding whether a prolonged loss should reset the integrator—is part of the next validation stage.

6.2. Results

Simulation. In Gazebo the CenterOnTarget PI controller successfully kept the simulated camera pointed at the target through the MAVLink backend. This indicated that the control logic, the error normalization, and the PI gains were reasonable in the absence of hardware effects.

Figures 5 and 6 show the normalized horizontal error e_x and the commanded yaw rate for a static and a dynamic target, respectively. With a static target (Figure 5), a large initial offset ($e_x = -0.90$) is driven into the deadband within 2.31 s; a second target switch around $t = 40$ s produces a smaller transient that settles in 2.05 s. With a dynamic target (Figure 6), the controller tracks sustained motion across the full field of view, including a full-range excursion ($e_x \approx -1.0$ near $t = 44$ s), and returns towards the deadband each time the target slows. In both cases the commanded yaw rate follows the sign and magnitude of the error throughout, consistent with the intended PI behavior in simulation.

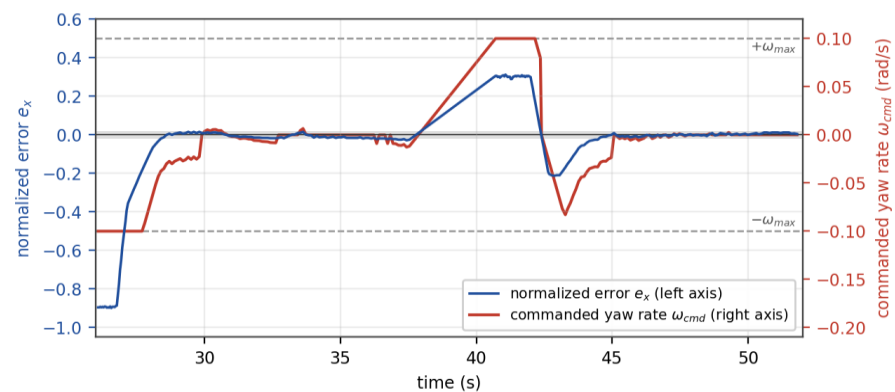


Figure 5. Simulation with static targets: normalized horizontal error e_x (blue, left axis) and commanded yaw rate (red, right axis) over time. Starting with large initial offset and a subsequent target switch with fast convergence.

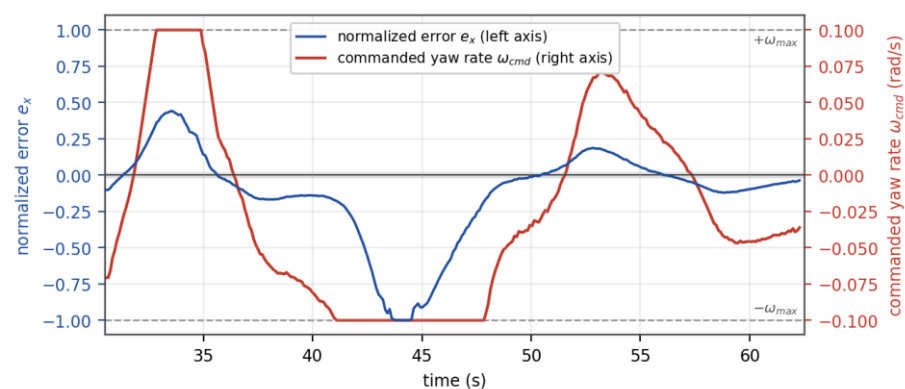


Figure 6. Simulation with a dynamic target: normalized horizontal error e_x (blue, left axis) and commanded yaw rate (red, right axis). The controller tracks continuous target motion, demonstrating bounded closed-loop behavior under the tested target motion.

Figure 7 shows a representative frame from one of these simulation runs: the NanoTrack bounding box center is rendered as a green “Tracking” marker on the Gazebo camera feed, and the bottom overlay reports the live controller state and commanded yaw rate.

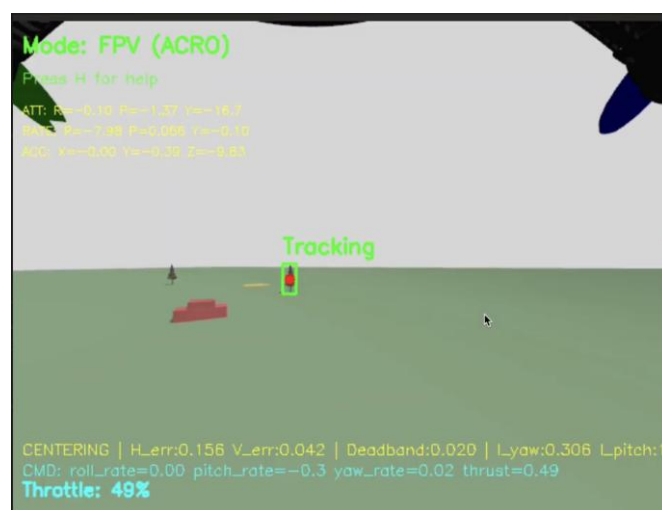


Figure 7. Representative frame from the Gazebo simulation. The green “Tracking” marker indicates the target bounding box.

Dry run. The dry run on the Raspberry Pi with the PrintCommandsFC backend reproduced the simulation behavior. Tracking worked as expected, control commands were generated at the expected rate, and no hardware errors or crashes were observed on the RPi side.

Bench test (propellers off). Before running with propellers, the full MSP integration was verified with propellers removed: correct commands were confirmed to reach the flight controller and MSP Override was exercised end-to-end without producing thrust. This step caught and resolved several integration issues that are not visible in simulation, ensuring that the system was safe to proceed to the propellers-on stand test.

Test stand (propellers on). Finally, with the drone placed on a stand that allowed free rotation in yaw (with pitch and roll restricted to small ranges), the closed-loop yaw tracking was demonstrated end-to-end. The drone physically rotated to center on a hand-moved target (Figure 4); the direction of rotation was correct in both left and right cases, and the behavior was observed across three runs with the target moved in both directions. These runs used an earlier configuration ($K_p = 0.4$, $K_i = 0.3$, $\omega_{\max} = 0.6$ rad/s, and with neither the anti-windup clamp, the rate-of-change limiter, nor the immediate integrator reset at the deadband), and exhibited occasional overshoot: the drone rotated past the target and then corrected back to center. They were observed directly rather than recorded, as the controller data log was added afterwards. Lowering K_i and $\omega_{\max}$ to the values in Table 1 and adding the integral clamp and rate-of-change limiter—the proportional gain is unchanged—appeared to reduce the overshoot observed on the stand, although it is still sometimes present. Two further stand runs were then recorded with the controller data log; these are the runs analyzed in Section 6.3.

Figure 8 shows a typical frame captured during one of the test-stand runs: NanoTrack has acquired a red book as the target and the green bounding box tracks it in real time, while the bottom HUD confirms the CENTERING state and the current controller output. No loss of track occurred on this high-contrast target in the two data-logged runs; low-contrast targets (plain walls, uniform surfaces) were found to cause tracker drift and are left for future work.



Figure 8. Representative frame from the yaw-only test-stand runs. The green bounding box marks the tracking region around a target object.

Figure 9 shows the error and commanded yaw-rate signals from one of the test-stand runs with a hand-moved target. Compared to simulation, the signal is noisier, due to mechanical vibration, the drone not being held perfectly still, extra USB latency, and lower video feed quality. Nevertheless, the commanded yaw rate closely follows the error signal throughout the run, and the error converges toward zero as the target slows after $t \approx 85$ s. During $t = 55\text{--}75$ s the target is moved through a series of displacements; the controller responds to each in turn, with the commanded yaw rate tracking the sign of e_x and the error returning toward zero between moves, consistent with the intended PI behavior under repeated target repositioning.

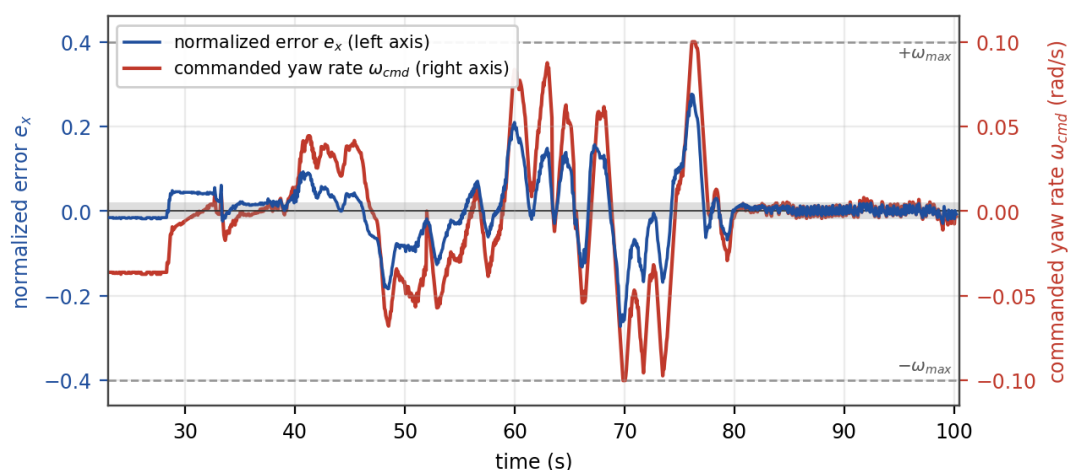


Figure 9. Real hardware on yaw-only test stand with a hand-moved dynamic target: normalized horizontal error e_x (blue, left axis) and commanded yaw rate (red, right axis). Higher noise relative to simulation reflects mechanical vibration, and tracker jitter; the commanded yaw rate nonetheless tracks the error, and the error returns towards the deadband once the target stops moving.

6.3. Results Analysis

The preceding results establish that the loop closes end to end on a real airframe. This subsection examines that behavior in more detail, using the controller logs recorded during the runs already described; no additional experiments were performed. The emphasis is on what the recorded signals reveal about the internal behavior of the controller, rather

than on aggregate performance figures: the runs differ in target, in operator movement and in duration, so summary error statistics would reflect the scenarios as much as the controller, and no baseline implementation was available to compare them against.

Definitions. All quantities are computed from the controller log over samples with a valid track. A displacement event begins (onset) at the first sample with $|e_x| \geq 0.10$ after the error has been below 0.03; its evaluation window extends to the next onset, so that a subsequent target movement is never attributed to the controller. The settling time is the time from onset to the first sample after which $|e_x|$ remains inside the deadband ($|e_x| < \varepsilon = 0.02$) for at least 0.4 s within the window; an event that does not meet this condition within its window is counted as not settled. The overshoot is the largest excursion of e_x with the sign opposite to that at onset, after the first sign change within the window, expressed as a percentage of the peak $|e_x|$ in the window. The mean absolute error is the mean of $|e_x|$ over all samples with a valid track.

6.3.1. Decomposition of a Single Response

Figure 10 decomposes one hardware response into the two terms of the control law and the command finally issued, which exposes the behavior of all three limiters described in Section 4 within a single event.

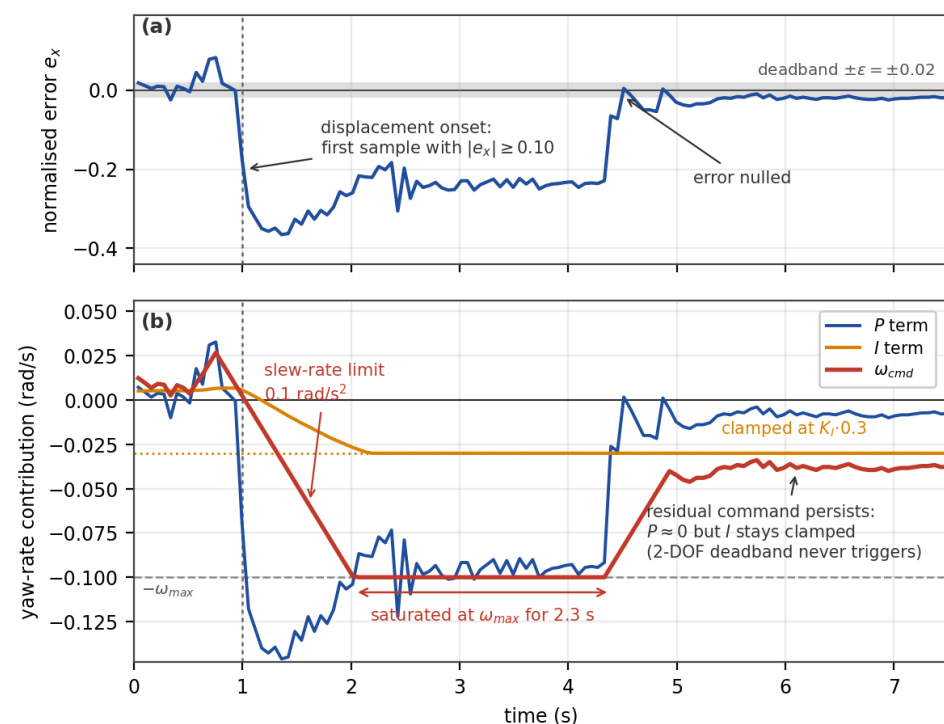


Figure 10. Decomposition of the yaw controller's response to a single target displacement on the yaw-only test stand ($K_P = 0.4$, $K_I = 0.1$, $\omega_{max} = 0.1$ rad/s). The dashed line marks the first sample at which $|e_x| \geq 0.10$; the target was displaced by hand over approximately 0.3 s around that instant, so the input is a ramp-and-hold rather than a step. (a) Normalized horizontal error, with the $\pm \varepsilon$ deadband shaded. (b) The proportional and integral contributions and the commanded yaw rate.

Three effects are visible in sequence. The proportional term responds immediately to the displacement and reaches 0.146 rad/s, approximately 1.5 times the configured rate limit, but the issued command does not follow it: it descends linearly, at a measured 0.101 rad/s² against a configured rate-of-change limit of 0.1 rad/s², and requires roughly 0.9 s to reach the limit. It then remains at ω_{max} for 2.3 s, during which the proportional term alone continues to request between 0.09 and 0.13 rad/s; over this interval the rate limit, rather

than the gain, determines the response. The integral term meanwhile accumulates and settles at exactly 0.03 rad/s, the product of K_I and the accumulator clamp, and remains there: the anti-windup treatment is active for a substantial part of the response rather than being a precaution that never engages.

6.3.2. Effect of the Anti-Windup and Rate-Limiting Changes

The configuration described in Section 4 was reached after the earlier one described above, which lacked the anti-windup clamp, the rate-of-change limiter and the immediate integrator reset. The stand runs under that earlier configuration predate the controller data log, so no hardware comparison is possible from the recorded data; both configurations were, however, exercised in simulation and logged, which permits a quantitative comparison there. Under the earlier configuration the error does not reach the deadband: after a target displacement it comes to rest at an offset of roughly 0.17 and remains there until the target is moved again. Pooled over all logged simulation runs, one of nineteen displacement events settled under the earlier configuration against fourteen of twenty-five under the present one, the median overshoot fell from 12.5 % to 5.2 %, and the mean absolute error from 0.309 to 0.175. This is the behavior that motivated the changes: the earlier controller did not oscillate but came to rest at a persistent offset, which is the expected consequence of an unbounded integral term combined with a reset condition that never fires. The comparison is drawn between the runs available at each configuration rather than between repeated trials of a single scenario, so it characterizes the two configurations as they were exercised and is not a controlled experiment. It is therefore exploratory: differences may also reflect the different target motions in the runs and cannot be solely attributed to the controller changes.

6.3.3. Tracker Reliability

Across the two data-logged stand runs, comprising 111 s of continuous closed-loop operation, the tracker reported a valid target on every sample and no loss of track occurred. This result is specific to the high-contrast target adopted for these experiments, and should be read together with the observation reported above that low-contrast targets cause drift. In the simulation runs, by contrast, five loss episodes were recorded, of between 2.5 and 7.3 s. A deliberate evaluation of loss-of-track behavior, including recovery after occlusion, was not performed and remains future work.

7. Discussion and Limitations

The results of Section 6 are those of a proof of concept. They show that the integrated system—tracker, controller, flight-stack abstraction and MSP Override command path—closes the visual loop on a real Betaflight airframe, and they document how the controller behaves in the recorded runs. They are not a performance characterization of the controller. This section interprets the results and states the limitations that bound them; each limitation is matched by an item of the validation protocol in Section 8.

Interpretation of the controller logs. The decomposition in Section 6.3.1 shows that, in the tested configuration, the response to a large displacement is shaped by the limiters rather than by the gains: the rate-of-change limiter sets the onset of the response, the yaw-rate cap bounds it during large displacements, and the anti-windup clamp is active during ordinary operation rather than only as a safeguard. The comparison in Section 6.3.2 indicates why these elements were introduced, but it is drawn between the simulation runs available for each configuration and is not a controlled comparison.

Validation scope. Closed-loop control acts on one axis (yaw), with the airframe on a test stand that restricts pitch and roll and prevents translation, indoors, and with hand-

moved high-contrast targets. Two stand runs were data-logged (111 s of closed-loop operation); the three earlier runs were observed but not recorded and used a different controller configuration. Because the hand-moved target motion differs between runs, the recorded runs do not support aggregate statistics, reproducibility claims or a comparison between controller variants, and none are made. The stand results demonstrate that the integrated system operates end to end on a real airframe; they do not characterize free-flight dynamics, which involve coupled attitude, translation and aerodynamic effects that the stand removes.

Controller as implemented. As described in Section 4.2, the deadband condition that zeroes the output and resets the integrator is evaluated on both image axes. In the yaw-only experiments this branch was effectively inactive on hardware, so a bounded residual yaw command could persist while the target was horizontally centered. The results are therefore reported for the controller as implemented. Per-axis gating is the correct design and will be used in the next validation stage.

Physical response of the airframe. The plots in Section 6 compare the image error with the commanded yaw rate. The physical rotation of the airframe was confirmed by direct observation of every stand run (Figure 4), and it is reflected in the image error, which decreases as the camera turns towards the target; with a hand-moved target, however, the image error alone cannot separate the rotation of the airframe from the motion of the target. The rotation was not measured with an independent sensor. Measurement of the actual yaw rate and angle is part of the validation protocol.

Tracker. NanoTrack is used as an existing building block; its selection and benchmarking on the Raspberry Pi 5 were reported in [11]. Its robustness to low contrast, clutter, occlusion, illumination changes and prolonged target loss was not evaluated in this study, and the high-contrast targets used on the stand were chosen so that tracker failures would not mask the behavior of the control loop. On loss of track the controller issues a zero yaw command and freezes the integral state (Section 6.1); the pilot can take back control at any time with the override switch. An explicit timeout that resets the integrator after a prolonged loss is part of the next stage.

8. Conclusions and Future Work

We have presented a proof-of-concept closed-loop visual servoing system for FPV drones running Betaflight, built around a FlightController abstraction that hides the difference between the simulation stack (ArduPilot SITL, Gazebo, MAVLink) and the real hardware stack (Betaflight, MSP Override). The same control code and the same binary were used at every stage: first in Gazebo, then in a dry run on the Raspberry Pi 5, then on a bench with the propellers removed, and finally on a real airframe mounted on a yaw-only test stand. Only command-line flags changed between stages. On the stand the drone rotates to keep a hand-moved target centered in the image, in the correct direction on either side, and the behavior was observed in all five stand runs.

The architecture arrived at here is the one the recent literature converges on: a lightweight tracker feeding a classical regulator. The difference lies in the hardware it runs on. The systems reviewed in Section 2 pair a MAVLink autopilot with a GPU-class companion board, and they can rely on a position and velocity estimate provided by the flight stack; most of the systems run on proprietary platforms or with computation on a ground laptop. The companion computer used in this work is a CPU-only Raspberry Pi 5, and Betaflight provides no position or velocity estimate to close an outer loop against—only the attitude estimate used for its own stabilization. Under these conditions the model-based and learning-based methods have nothing to build on, and a PI regulator on the image error is a practical choice.

The staged validation allowed to fix the technical issues gradually. Each stage failed in its own way and exposed faults that simulation could not have revealed.

Analysis of the recorded controller logs supported several of the design decisions described in Section 4. Decomposing the commanded yaw rate into its proportional and integral contributions showed that the anti-windup clamp binds during ordinary operation rather than acting as a safeguard that never engages, so it is an active element of the control law. The same decomposition showed that the rate-of-change limiter, and not the proportional gain, shapes the onset of each response, and that the conservative yaw-rate cap does become the binding constraint during a large displacement; the target was nonetheless re-centered without loss of track, which suggests that, for the target motions tested, the cap did not prevent re-centering.

The next stage of this research follows from the limitations stated in Section 7 and is organized as a validation protocol:

- Repeatable target motion: a scripted target, for example displayed on a screen, generating predefined move trajectories, so that runs can be repeated under identical conditions.
- Controlled comparison: P-only, PI and PI with anti-windup and rate limiting evaluated under identical target trajectories, with repeated trials, reporting settling time, overshoot, steady-state error, tracking success rate and their variability.
- Controller corrections: per-axis deadband gating, and an integrator reset after a prolonged loss of track.
- Independent measurement of the physical response: yaw rate and angle from the flight-controller log or an external angle reference, and experimental validation of the PWM-to-yaw-rate mapping.
- Timing and resources: measurement of end-to-end latency, per-stage processing time, command thread timing, frame drops, CPU and memory utilization on the companion computer.
- Hardware link: replacement of USB connection with a hardware UART and measuring its impact on latency.
- Tracker robustness: evaluation under low contrast, occlusion, illumination changes, fine tuning for the natural scenes relevant to ecology applications where required.
- Free flight: extension of the loop beyond yaw, applying the same regulator to the vertical image error and through it to pitch and throttle, so that the drone holds a distance to a target.

Field trial: a trial with a live ecological target, the target application of this work.

Author Contributions: Conceptualization, M.D., K.V. and A.M.; methodology, M.D., K.V. and A.M.; software, K.V. and P.P.; validation, K.V., A.M. and P.P.; investigation, M.D., K.V. and A.M.; writing—original draft preparation, M.D., K.V., A.M. and P.P.; writing—review and editing, M.D., A.M., K.V. and P.P.; funding acquisition, P.P. All authors have read and agreed to the published version of the manuscript.

Funding: Financed by a subvention from the Ministry of Science and Higher Education, Poland to the Silesian University of Technology. This research was supported by the National Research Foundation of Ukraine under project No. 2025.07/0207, “Concept of a Knowledge-Oriented Technology for Modeling and Forecasting the State of the Natural Environment Based on Imprecise Data.”

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data cited in this manuscript are available from the published papers or the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Manfreda, S.; McCabe, M.F.; Miller, P.E.; Lucas, R.; Pajuelo Madrigal, V.; Mallinis, G.; Ben Dor, E.; Helman, D.; Estes, L.; Ciraolo, G.; et al. On the Use of Unmanned Aerial Systems for Environmental Monitoring. *Remote Sens.* **2018**, *10*, 641. <https://doi.org/10.3390/rs10040641>.
2. Hodgson, J.C.; Mott, R.; Baylis, S.M.; Pham, T.T.; Wotherspoon, S.; Kilpatrick, A.D.; Raja Segaran, R.; Reid, I.; Terauds, A.; Koh, L.P. Drones count wildlife more accurately and precisely than humans. *Methods Ecol. Evol.* **2018**, *9*, 1160–1167. <https://doi.org/10.1111/2041-210X.12974>.
3. Schad, L.; Fischer, J. Opportunities and risks in the use of drones for studying animal behaviour. *Methods Ecol. Evol.* **2023**, *14*, 1864–1872. <https://doi.org/10.1111/2041-210X.13922>.
4. Betaflight Flight Controller Firmware. Available online: <https://github.com/betaflight/betaflight> (accessed on 24 August 2026).
5. Modes—Flight Mode Reference (MSP OVERRIDE, Mode ID 50). Betaflight Documentation. Available online: <https://betaflight.com/docs/wiki/guides/current/Modes> (accessed on 24 August 2026).
6. SITL Simulator (Software in the Loop). ArduPilot Documentation. Available online: <https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html> (accessed on 24 August 2026).
7. Furrer, F.; Burri, M.; Achtelik, M.; Siegwart, R. RotorS—A Modular Gazebo MAV Simulator Framework. In *Robot Operating System (ROS): The Complete Reference*; Koubaa, A., Ed.; Springer International Publishing: Cham, Switzerland, 2016; Volume 625, pp. 595–625. https://doi.org/10.1007/978-3-319-26054-9_23.
8. Foehn, P.; Kaufmann, E.; Romero, A.; Penicka, R.; Sun, S.; Bauersfeld, L.; Laengle, T.; Cioffi, G.; Song, Y.; Loquercio, A.; et al. Agilicious: Open-source and open-hardware agile quadrotor for vision-based flight. *Sci. Robot.* **2022**, *7*, eabl6259. <https://doi.org/10.1126/scirobotics.abl6259>.
9. Panerati, J.; Sajjadi, S.; Soleymanpour, S.; Mehta, V.; Mantegh, I. aerial-autonomy-stack: A faster-than-real-time, autopilot-agnostic, ROS2 framework to simulate and deploy perception-based drones. *arXiv* **2026**, arXiv:2602.07264.
10. NanoTrack: A Lightweight Deep-Learning-Based Object Tracker. Available online: <https://github.com/HonglinChu/NanoTrack> (accessed on 24 August 2026).
11. Verhun, K.; Dyvak, M. Visual Object Tracking using Neural Networks for Embedded Systems. In *Proceedings of the 2025 15th International Conference on Advanced Computer Information Technologies (ACIT)*, Šibenik, Croatia, 17–19 September 2025; IEEE: New York, NY, USA, 2025; pp. 836–840. <https://doi.org/10.1109/ACIT65614.2025.11185744>.
12. Zhang, Z.; Zhu, L. A Review on Unmanned Aerial Vehicle Remote Sensing: Platforms, Sensors, Data Processing Methods, and Applications. *Drones* **2023**, *7*, 398. <https://doi.org/10.3390/drones7060398>.
13. Jones, G.P., IV; Pearlstine, L.G.; Percival, H.F. An assessment of small unmanned aerial vehicles for wildlife research. *Wildl. Soc. Bull.* **2006**, *34*, 750–758. [https://doi.org/10.2193/0091-7648\(2006\)34\[750:AAOSUA\]2.0.CO;2](https://doi.org/10.2193/0091-7648(2006)34[750:AAOSUA]2.0.CO;2).
14. Watts, A.C.; Perry, J.H.; Smith, S.E.; Burgess, M.A.; Wilkinson, B.E.; Szantoi, Z.; Ifju, P.G.; Percival, H.F. Small unmanned aircraft systems for low-altitude aerial surveys. *J. Wildl. Manag.* **2010**, *74*, 1614–1619. <https://doi.org/10.2193/2009-425>.
15. Koh, L.P.; Wich, S.A. Dawn of drone ecology: Low-cost autonomous aerial vehicles for conservation. *Trop. Conserv. Sci.* **2012**, *5*, 121–132. <https://doi.org/10.1177/194008291200500202>.
16. Anderson, K.; Gaston, K.J. Lightweight unmanned aerial vehicles will revolutionize spatial ecology. *Front. Ecol. Environ.* **2013**, *11*, 138–146. <https://doi.org/10.1890/120150>.
17. Anderson, K.; Gonzalez, F.; Gaston, K.J. Drones in ecology: Ten years back and forth. *BioScience* **2025**, *75*, 664–680. <https://doi.org/10.1093/biosci/biaf069>.
18. Ecke, S.; Dempewolf, J.; Frey, J.; Schwaller, A.; Endres, E.; Klemmt, H.-J.; Tiede, D.; Seifert, T. UAV-Based Forest Health Monitoring: A Systematic Review. *Remote Sens.* **2022**, *14*, 3205. <https://doi.org/10.3390/rs14133205>.
19. Shahi, T.B.; Xu, C.-Y.; Neupane, A.; Guo, W. Recent Advances in Crop Disease Detection Using UAV and Deep Learning Techniques. *Remote Sens.* **2023**, *15*, 2450. <https://doi.org/10.3390/rs15092450>.
20. Wasehun, E.T.; Hashemi Beni, L.; Di Vittorio, C.A. UAV and satellite remote sensing for inland water quality assessments: A literature review. *Environ. Monit. Assess.* **2024**, *196*, 277. <https://doi.org/10.1007/s10661-024-12342-6>.
21. Jessin, J.; Heinzlef, C.; Long, N.; Serre, D. A Systematic Review of UAVs for Island Coastal Environment and Risk Monitoring: Towards a Resilience Assessment. *Drones* **2023**, *7*, 206. <https://doi.org/10.3390/drones7030206>.
22. Duangsuwan, S.; Prapruetdee, P. Drone-Enabled AI Edge Computing and 5G Communication Network for Real-Time Coastal Litter Detection. *Drones* **2024**, *8*, 750. <https://doi.org/10.3390/drones8120750>.

23. Fosco, D.; De Molfetta, M.; Renzulli, P.; Notarnicola, B. Progress in monitoring methane emissions from landfills using drones: An overview of the last ten years. *Sci. Total Environ.* **2024**, *945*, 173981. <https://doi.org/10.1016/j.scitotenv.2024.173981>.
24. Khan, A.; Gupta, S.; Gupta, S.K. Emerging UAV technology for disaster detection, mitigation, response, and preparedness. *J. Field Robot.* **2022**, *39*, 905–955. <https://doi.org/10.1002/rob.22075>.
25. Shamta, I.; Demir, B.E. Development of a deep learning-based surveillance system for forest fire detection and monitoring using UAV. *PLoS ONE* **2024**, *19*, e0299058. <https://doi.org/10.1371/journal.pone.0299058>.
26. Hutsul, T.; Khobzei, M.; Tkach, V.; Krulikovskyi, O.; Moisiuk, O.; Ivashko, V.; Samila, A. Review of approaches to the use of unmanned aerial vehicles, remote sensing and geographic information systems in humanitarian demining: Ukrainian case. *Heliyon* **2024**, *10*, e29142. <https://doi.org/10.1016/j.heliyon.2024.e29142>.
27. Aliane, N. Drones and AI-Driven Solutions for Wildlife Monitoring. *Drones* **2025**, *9*, 455. <https://doi.org/10.3390/drones9070455>.
28. Axford, D.; Soheli, F.; Vanderklift, M.A.; Hodgson, A.J. Collectively advancing deep learning for animal detection in drone imagery: Successes, challenges, and research gaps. *Ecol. Inform.* **2024**, *83*, 102842. <https://doi.org/10.1016/j.ecoinf.2024.102842>.
29. Kline, J.; Afridi, S.; Rolland, E.G.A.; Maalouf, G.; Laporte-Devyllder, L.; Stewart, C.; Crofoot, M.; Stewart, C.V.; Rubenstein, D.I.; Berger-Wolf, T. Studying collective animal behaviour with drones and computer vision. *Methods Ecol. Evol.* **2025**, *16*, 2229–2259. <https://doi.org/10.1111/2041-210X.70128>.
30. Kline, J.; Zhong, A.; Irizarry, K.; Stewart, C.V.; Stewart, C.; Rubenstein, D.I.; Berger-Wolf, T. WildWing: An open-source, autonomous and affordable UAS for animal behaviour video monitoring. *Methods Ecol. Evol.* **2026**, *17*, 291–300. <https://doi.org/10.1111/2041-210X.70018>.
31. Afridi, S.; Laporte-Devyllder, L.; Maalouf, G.; Kline, J.M.; Penny, S.G.; Hlebowicz, K.; Cawthorne, D.; Schultz Lundquist, U.P. Impact of Drone Disturbances on Wildlife: A Review. *Drones* **2025**, *9*, 311. <https://doi.org/10.3390/drones9040311>.
32. Duporge, I.; Spiegel, M.P.; Thomson, E.R.; Chapman, T.; Lamberth, C.; Pond, C.; Macdonald, D.W.; Wang, T.; Klinck, H. Determination of optimal flight altitude to minimise acoustic drone disturbance to wildlife using species audiograms. *Methods Ecol. Evol.* **2021**, *12*, 2196–2207. <https://doi.org/10.1111/2041-210X.13691>.
33. Koubaa, A.; Ammar, A.; Abdelkader, M.; Alhabashi, Y.; Ghouti, L. AERO: AI-Enabled Remote Sensing Observation with Onboard Edge Computing in UAVs. *Remote Sens.* **2023**, *15*, 1873. <https://doi.org/10.3390/rs15071873>.
34. Iglay, R.B.; Jones, L.R.; Elmore, J.A.; Evans, K.O.; Samiappan, S.; Pfeiffer, M.B.; Blackwell, B.F. Wildlife monitoring with drones: A survey of end users. *Wildl. Soc. Bull.* **2024**, *48*, e1533. <https://doi.org/10.1002/wsb.1533>.
35. Idrissi, M.; Salami, M.R.; Annaz, F. A Review of Quadrotor Unmanned Aerial Vehicles: Applications, Architectural Design and Control Algorithms. *J. Intell. Robot. Syst.* **2022**, *104*, 22. <https://doi.org/10.1007/s10846-021-01527-7>.
36. Khalid, A.; Mushtaq, Z.; Arif, S.; Zeb, K.; Khan, M.A.; Bakshi, S. Control Schemes for Quadrotor UAV: Taxonomy and Survey. *ACM Comput. Surv.* **2024**, *56*, 1–32. <https://doi.org/10.1145/3617652>.
37. Nascimento, T.P.; Saska, M. Position and attitude control of multi-rotor aerial vehicles: A survey. *Annu. Rev. Control* **2019**, *48*, 129–146. <https://doi.org/10.1016/j.arcontrol.2019.08.004>.
38. Lopez-Sanchez, I.; Moreno-Valenzuela, J. PID control of quadrotor UAVs: A survey. *Annu. Rev. Control* **2023**, *56*, 100900. <https://doi.org/10.1016/j.arcontrol.2023.100900>.
39. Lopez-Franco, C.; Gomez-Avila, J.; Alanis, A.Y.; Arana-Daniel, N.; Villaseñor, C. Visual Servoing for an Autonomous Hexarotor Using a Neural Network Based PID Controller. *Sensors* **2017**, *17*, 1865. <https://doi.org/10.3390/s17081865>.
40. Rinaldi, M.; Primatesta, S.; Guglieri, G. A Comparative Study for Control of Quadrotor UAVs. *Appl. Sci.* **2023**, *13*, 3464. <https://doi.org/10.3390/app13063464>.
41. Muluaem, Y.L.; Jin, G.G.; Kwon, J.; Ahn, J. Backstepping Sliding Mode Control of Quadrotor UAV Trajectory. *Mathematics* **2025**, *13*, 3205. <https://doi.org/10.3390/math13193205>.
42. Sun, S.; Romero, A.; Foehn, P.; Kaufmann, E.; Scaramuzza, D. A Comparative Study of Nonlinear MPC and Differential-Flatness-Based Control for Quadrotor Agile Flight. *IEEE Trans. Robot.* **2022**, *38*, 3357–3373. <https://doi.org/10.1109/TRO.2022.3177279>.
43. Kaufmann, E.; Bauersfeld, L.; Loquercio, A.; Müller, M.; Koltun, V.; Scaramuzza, D. Champion-level drone racing using deep reinforcement learning. *Nature* **2023**, *620*, 982–987. <https://doi.org/10.1038/s41586-023-06419-4>.
44. Sönmez, S.; Rutherford, M.; Valavanis, K. A Survey of Offline- and Online-Learning-Based Algorithms for Multirotor UAVs. *Drones* **2024**, *8*, 116. <https://doi.org/10.3390/drones8040116>.
45. Hutchinson, S.; Hager, G.D.; Corke, P.I. A tutorial on visual servo control. *IEEE Trans. Robot. Autom.* **1996**, *12*, 651–670. <https://doi.org/10.1109/70.538972>.
46. Chaumette, F.; Hutchinson, S. Visual servo control, Part I: Basic approaches. *IEEE Robot. Autom. Mag.* **2006**, *13*, 82–90. <https://doi.org/10.1109/MRA.2006.250573>.

47. Machkour, Z.; Ortiz-Arroyo, D.; Durdevic, P. Classical and Deep Learning based Visual Servoing Systems: A Survey on State of the Art. *J. Intell. Robot. Syst.* **2022**, *104*, 11. <https://doi.org/10.1007/s10846-021-01540-w>.
48. Cong, V.D.; Hanh, L.D. A review and performance comparison of visual servoing controls. *Int. J. Intell. Robot. Appl.* **2023**, *7*, 65–90. <https://doi.org/10.1007/s41315-023-00270-6>.
49. Arafat, M.Y.; Alam, M.M.; Moh, S. Vision-Based Navigation Techniques for Unmanned Aerial Vehicles: Review and Challenges. *Drones* **2023**, *7*, 89. <https://doi.org/10.3390/drones7020089>.
50. Fu, G.; Chu, H.; Liu, L.; Fang, L.; Zhu, X. Deep Reinforcement Learning for the Visual Servoing Control of UAVs with FOV Constraint. *Drones* **2023**, *7*, 375. <https://doi.org/10.3390/drones7060375>.
51. Ferreira, D.; Basiri, M. Dynamic Target Tracking and Following with UAVs Using Multi-Target Information: Leveraging YOLOv8 and MOT Algorithms. *Drones* **2024**, *8*, 488. <https://doi.org/10.3390/drones8090488>.
52. Gharsa, O.; Touba, M.M.; Boumehraz, M.; Agram, N. Autonomous Vision-Based Object Detection and Tracking System for Quadrotor Unmanned Aerial Vehicles. *Sensors* **2025**, *25*, 6403. <https://doi.org/10.3390/s25206403>.
53. MAVSDK: MAVLink SDK for Drones. Available online: <https://github.com/mavlink/MAVSDK> (accessed on 24 August 2026).
54. Bosello, M.; Pinzarrone, F.; Kiade, S.; Aguiari, D.; Keuter, Y.; AlShehhi, A.; Caminati, G.; Wong, K.L.; Chou, K.S.; Halepota, J.; et al. On Your Own: Pro-Level Autonomous Drone Racing in Uninstrumented Arenas. *IEEE Robot. Autom. Lett.* **2026**, *11*, 2674–2681. <https://doi.org/10.1109/LRA.2026.3653405>.
55. TrackerNano (cv::TrackerNano), OpenCV 4.7+, Pull Request #22808. Available online: <https://github.com/opencv/opencv/pull/22808> (accessed on 24 August 2026).
56. Koenig, N.; Howard, A. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Sendai, Japan, 28 September–2 October 2004*; IEEE: New York, NY, USA, 2005; pp. 2149–2154. <https://doi.org/10.1109/IROS.2004.1389727>.
57. Raspberry Pi 5. Available online: <https://www.raspberrypi.com/products/raspberry-pi-5/> (accessed on 24 August 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.