

PID Color Tracking Gimbal Based on Arduino UNO Q

Tags

- OpenCV Color Recognition
- PID Closed-Loop Control
- Dual-Axis Gimbal
- Web Control
- HSV Color Space

Difficulty Level

Intermediate — This project involves computer vision (HSV color space and morphological processing), PID control algorithms (proportional-integral-derivative closed-loop), Arduino low-level PWM signal generation, and Web frontend interaction design. Beginners can complete basic tracking with guidance, but understanding PID parameter tuning and manual servo PWM generation requires a foundation in control theory and embedded development.

Project Cover Image

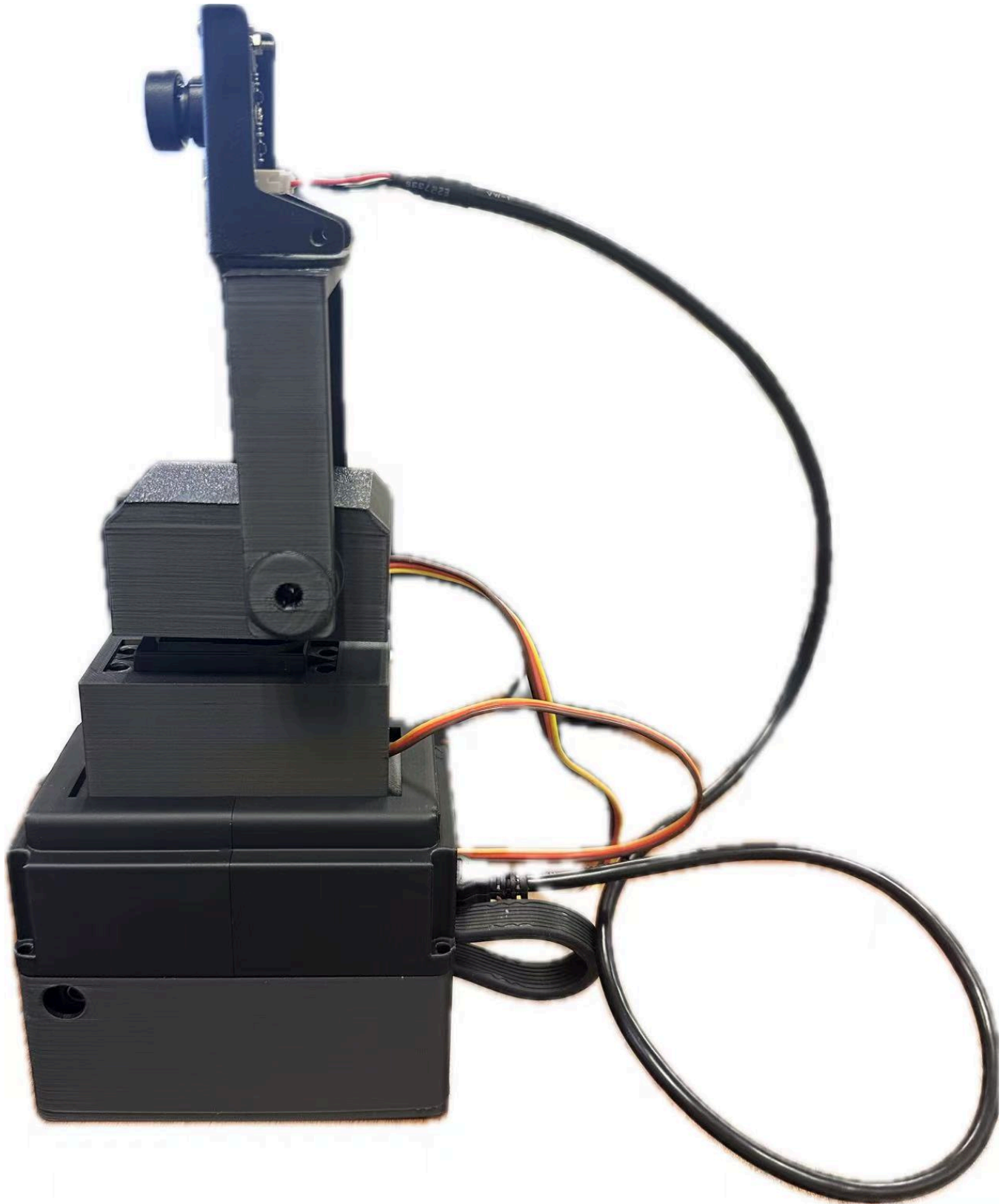


Figure 1: PID Color Tracking Gimbal Project Cover

Hardware Components

Building the PID color tracking gimbal requires the following key components. This is a pure gimbal project — no chassis or motors — the core goal is to make the camera "lock onto" a specified color target and follow it.

Gimbal Servos (2× MG996R Servos)

The two-axis gimbal is the project's actuator, giving the camera the ability to rotate horizontally and vertically:

- **Pan Servo (Horizontal Rotation):** Connected to digital pin D3, controls camera left-right rotation, range 0°–180°, center position 90°
- **Tilt Servo (Vertical Pitch):** Connected to digital pin D11, controls camera up-down angle, range 30°–150°, center position 90°

The MG996R is a high-torque metal gear servo with up to 9.4 kg·cm of torque, far exceeding the SG90's 1.8 kg·cm. Although larger and heavier, in a gimbal scenario that must support a USB camera and resist inertia, the MG996R's high torque and metal gear construction provide more stable and reliable support. This project does not use Arduino's Servo library, but instead **manually generates 50Hz PWM pulses** to drive the servos — an interesting low-level implementation choice.

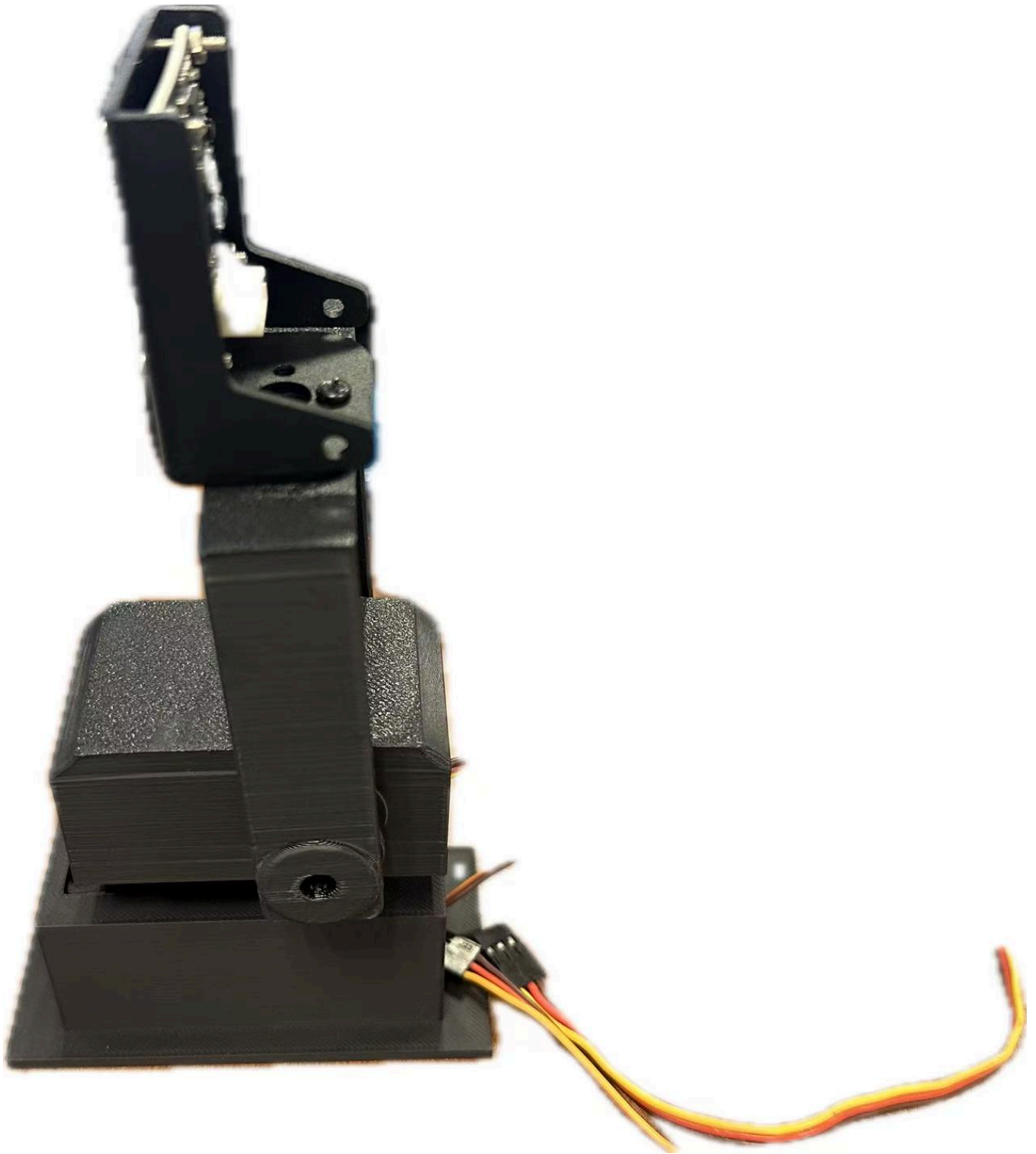


Figure 2: Dual-axis MG996R gimbal — the bottom Pan servo provides horizontal rotation, the top Tilt servo provides vertical pitch

Arduino UNO Q Development Board

The system's brain is the Arduino UNO Q — a modern Arduino platform based on STM32/Zephyr architecture:

- **Built-in WiFi:** Enables wireless communication and Web control
- **RouterBridge Communication Protocol:** Seamlessly connects the Python middleware with the Arduino firmware
- **Enhanced Processing Power:** Sufficient for handling real-time servo control commands

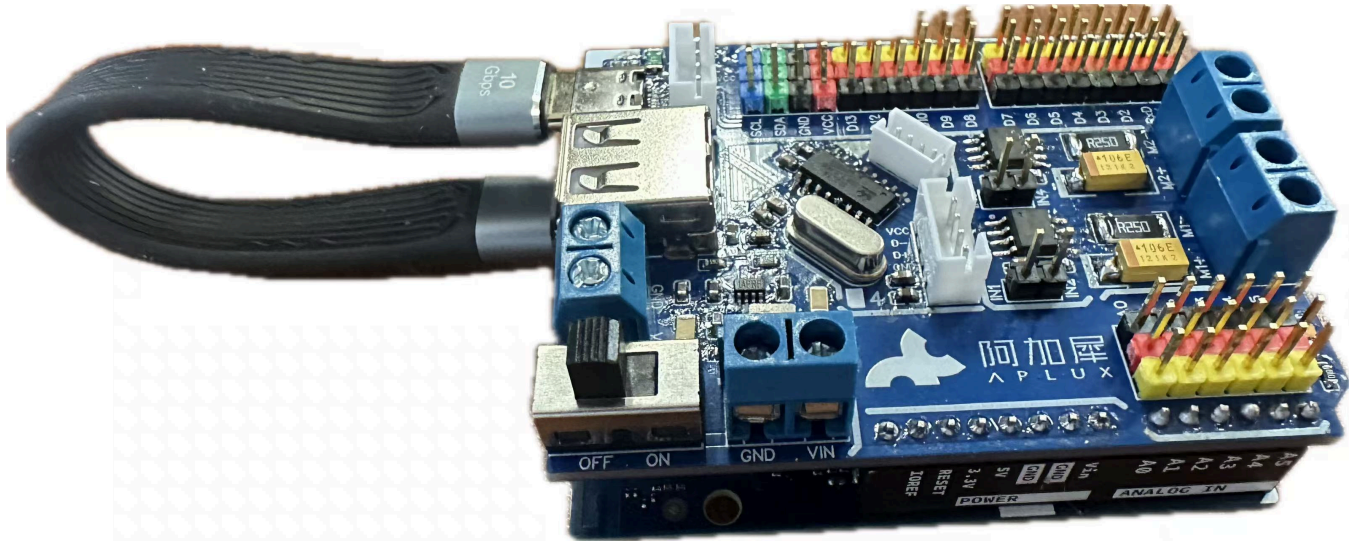


Figure 3: Arduino UNO Q connected to the expansion board via data cable

Expansion Board

The expansion board is a companion peripheral for the Arduino UNO Q, connected via a USB-C to C data cable, serving two key roles:

- **Power Distribution:** Converts and distributes battery power to the Arduino UNO Q and both MG996R servos, providing stable power supply
- **IO Expansion:** Provides additional interfaces and pin breakouts, making servo signal wire (D3, D11) connections cleaner and more convenient

The expansion board communicates with and powers the Arduino UNO Q through the USB-C data cable, eliminating the need for separate power wiring. The entire gimbal only needs servo signal wires and a single USB-C cable to complete all connections.

USB Camera

The camera is the gimbal's "eye", connected to the Arduino UNO Q via USB port:

- Provides 320×240 resolution real-time video stream
- Video frames are processed by Python-side OpenCV for color recognition and target positioning
- Processed frames are pushed to the Web frontend via Base64 encoding

Gimbal Mount

The mechanical mount supports the camera and two servos, forming a two-degree-of-freedom rotation mechanism:

- Pan servo is fixed to the base, providing horizontal rotation
- Tilt servo is mounted on the Pan servo's horn, providing vertical pitch
- Camera is fixed to the Tilt servo's horn



Figure 4: Gimbal base structure and power module

Complete Hardware List

Component	Qty	Model	Purpose
Gimbal Servos	2	MG996R	Horizontal rotation (Pan) and vertical pitch (Tilt)
Development Board	1	Arduino UNO Q	STM32/Zephyr-based main controller
Expansion Board	1	Custom	Power distribution and IO expansion
C-to-C Cable	1	USB-C	Connects expansion board to Arduino UNO Q
Camera	1	USB Camera	Video capture and color recognition
Gimbal Mount	1 set	Custom	Dual-axis rotation mechanism
Power Supply	1	Battery	System power

Tools and Machines

- **Personal Computer:** For programming and initial configuration
- **Screwdriver Set:** For gimbal assembly (mainly Phillips head)
- **3D Printer** (optional): For printing custom gimbal mounts

Development Platform

- **Arduino App Lab:** Web IDE for programming the Arduino UNO Q
- **Python 3.13+:** Backend vision processing and PID control

- **OpenCV:** Computer vision library for HSV color space processing
 - **Modern Web Browser:** Chrome, Firefox, or Edge for the control interface
-

Project Description

Introduction

Color tracking is one of the most classic introductory projects in computer vision — making a camera "stare at" an object of a certain color, following wherever the object moves. This project implements this concept as a complete closed-loop control system: from color recognition, error calculation, PID control to servo output, forming a complete visual servo closed-loop chain.

Unlike traditional color tracking projects, this system has five standout features:

First, PID closed-loop control. Traditional approaches often use Bang-Bang control (fixed step left when the target is on the left), resulting in jerky tracking with significant overshoot. This project implements a complete PID controller — the proportional term provides fast response, the integral term eliminates steady-state error, and the derivative term suppresses overshoot oscillation. Combined with dead zone design and integral anti-windup, tracking motion is smooth and natural.

Second, Web-based dynamic color selection. No code modification needed to switch tracking targets — the Web interface provides a Hue rainbow slider and 6 preset color blocks (yellow, red, green, blue, orange, purple). Dragging the slider instantly changes the target tracking color. The HSV color space makes color selection intuitive and robust against lighting changes.

Third, real-time PID tuning panel. The Web interface embeds a PID parameter tuning panel that allows real-time adjustment of Kp, Ki, Kd, and dead zone size during active tracking, with 4 preset configurations (Default / P-only / Conservative / Aggressive). This makes PID tuning visual and interactive, eliminating the need to repeatedly modify code and re-upload.

Fourth, manual PWM servo driving. The Arduino side does not use the Servo library, instead manually generating 50Hz PWM pulse signals to directly drive the MG996R servos. This approach lets you understand servo operation at the signal level — how angles map to pulse widths, and how pulse widths compose a PWM cycle.

Fifth, servo center calibration. The Web interface includes a dedicated servo calibration panel — after one-click centering to 90°, use $\pm 1^\circ/\pm 5^\circ$ fine-tune buttons to precisely locate the servo's physical center position before installing the horn. This makes assembly precise rather than relying on "feel", ensuring accurate repositioning every time the horn is removed and reinstalled.

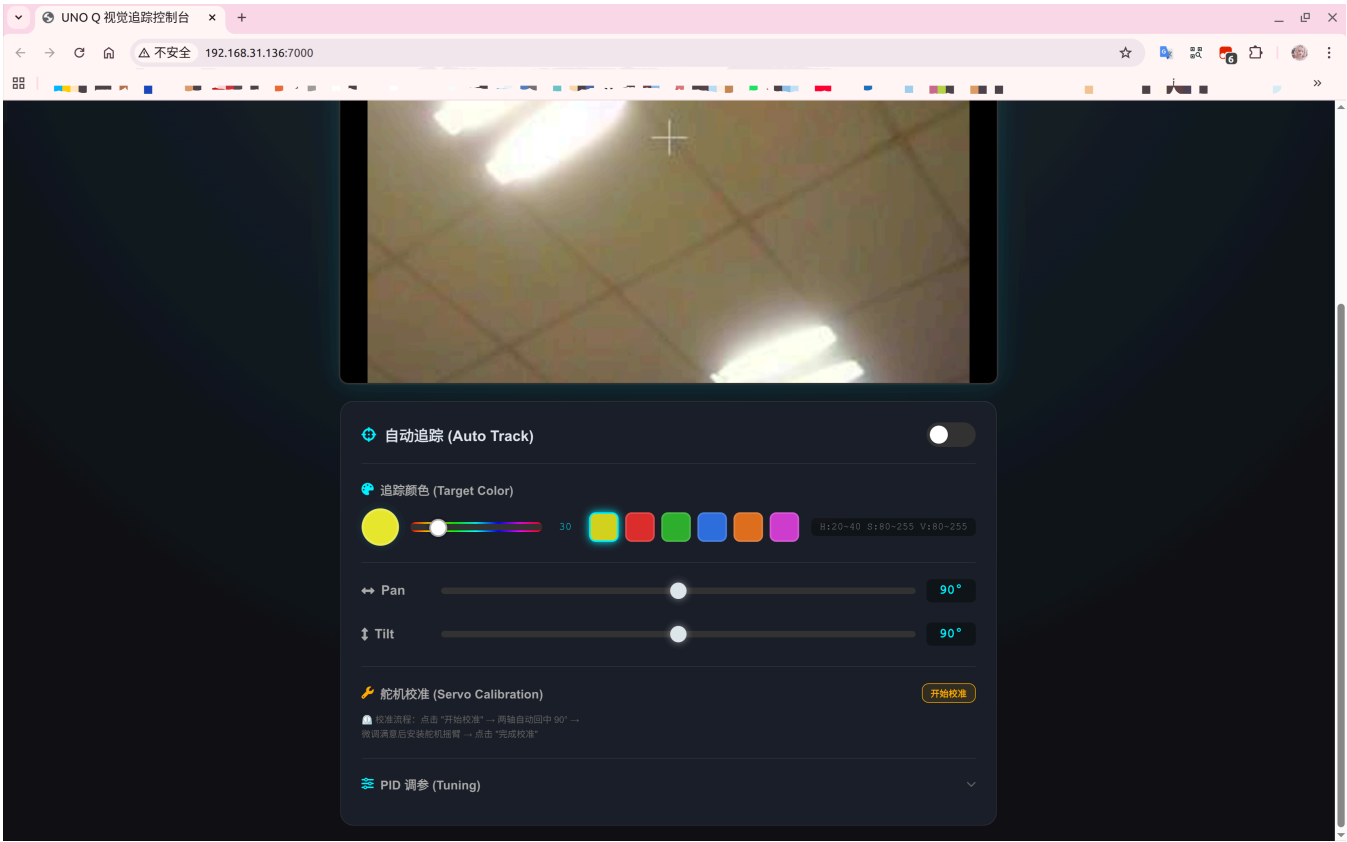


Figure 6: Web control panel — video stream (with tracking markers), Hue rainbow color bar, Pan/Tilt gimbal sliders, PID tuning panel, and servo calibration tools

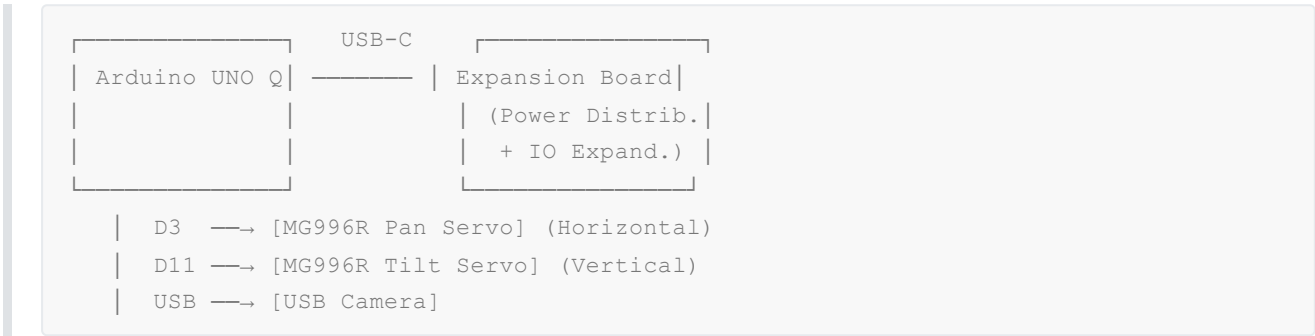
Circuit Diagram and Pin Configuration

The wiring for this project is very clean — two servos use only 2 digital pins, plus a USB-C data cable connecting the expansion board and USB camera.

Pin Assignment:

Function	Pin	Description
Gimbal Pan	D3	Camera horizontal rotation (0°–180°)
Gimbal Tilt	D11	Camera vertical pitch (30°–150°, center 90°)

Wiring Block Diagram:



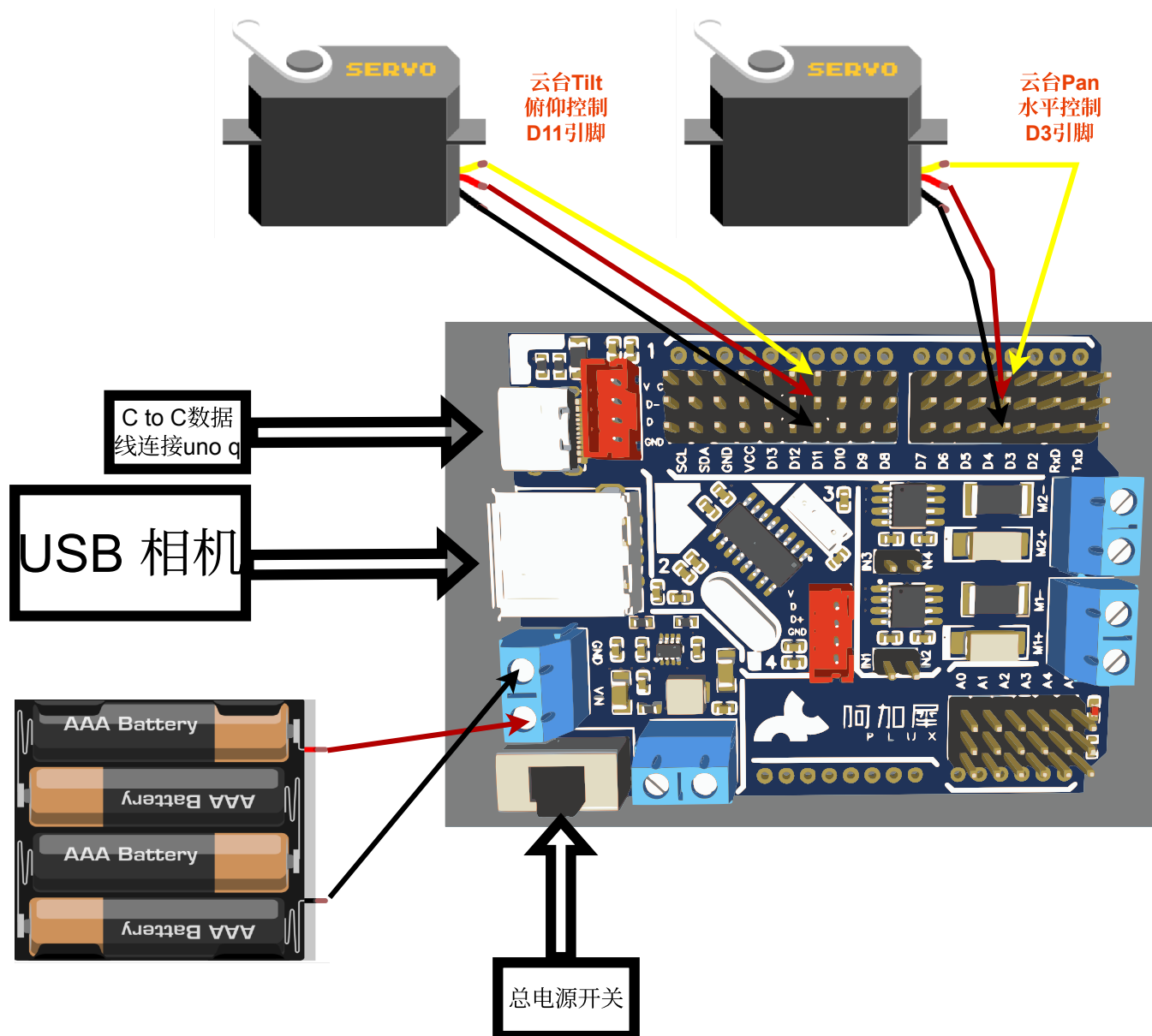
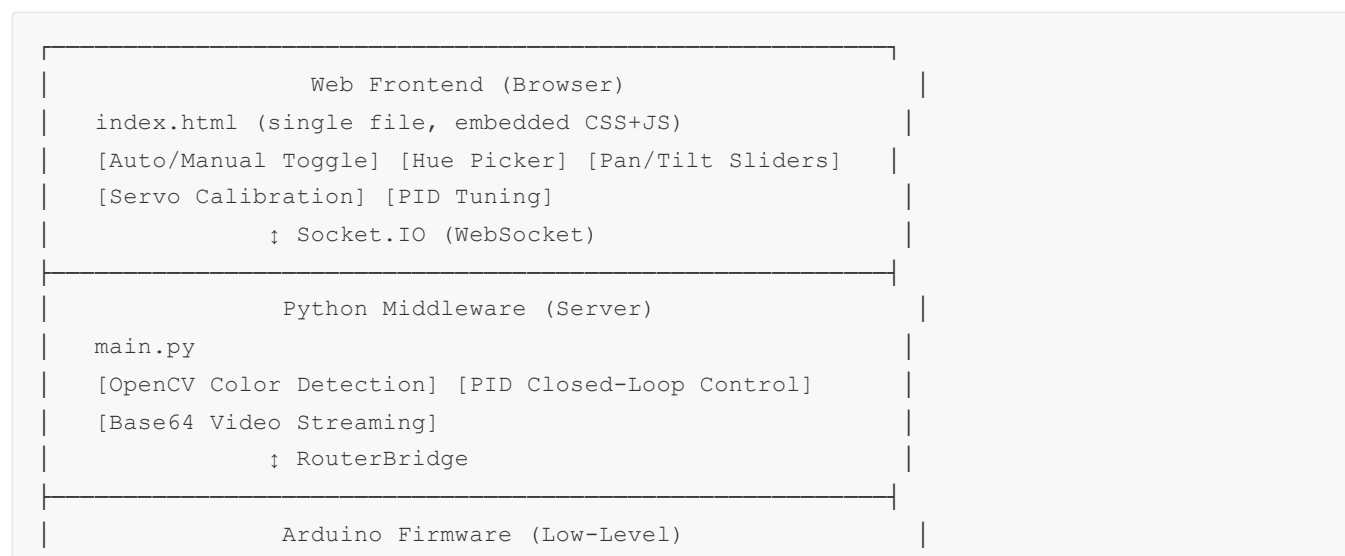


Figure 5: Module interconnection logic block diagram

System Architecture

The system uses a three-layer separated architecture, forming a complete visual servo closed-loop:



```
sketch.ino
[Manual 50Hz PWM Generation] [Angle→Pulse Mapping]
[D3 Pan Servo] [D11 Tilt Servo]
```

Closed-Loop Control Data Flow:

```
Camera Frame → OpenCV Color Detection → Target Centroid Coordinates
  → Pixel Error (target position - frame center)
  → PID Controller (error → angle correction)
  → Servo Angle Update
  → Bridge.call() → Arduino PWM Output
  → Servo Rotation → Camera Frame Changes (closed-loop)
```

Layer 1: Web Frontend (Browser)

The entire frontend is a single HTML file (with embedded CSS and JS), providing the following functional areas:

- **Mode Switch:** Auto-tracking / Manual control toggle
- **Color Selection Area:** Hue rainbow slider (0–180 mapping to OpenCV's H channel range) + 6 preset color blocks + real-time HSV range display
- **Gimbal Control:** Pan (horizontal) and Tilt (vertical) sliders, draggable in manual mode
- **Servo Calibration Panel:** One-click centering to 90° + ±1°/±5° fine-tuning, for precisely locating servo center during initial assembly or horn replacement
- **PID Tuning Panel:** Collapsible panel with Kp/Ki/Kd/Dead Zone sliders and 4 preset configurations
- **Video Stream Display:** Real-time display of OpenCV-processed frames (with tracking markers)

Layer 2: Python Middleware

The Python server is the system's core, handling three major responsibilities:

1. **Visual Processing:** OpenCV reads camera frames, converts to HSV color space, finds the target object's centroid coordinates through color thresholding and morphological processing
2. **PID Control:** Feeds the pixel offset between target centroid and frame center into the PID controller, calculates servo angle corrections, updates current angles, and sends commands to Arduino
3. **Video Streaming:** Encodes OpenCV-processed frames (with tracking circles and crosshairs) as JPEG, pushes to Web frontend via Base64 through Socket.IO

Layer 3: Arduino Firmware

Arduino runs lean C++ code with a single core task: **converting angle values to PWM pulse signals.**

- Does not use the Servo library — fully manual 50Hz PWM generation
- `angle_to_pulse()` function linearly maps 0°–180° to 500µs–2500µs pulse width
- `loop()` sequentially generates pulses for Pan and Tilt, then pads remaining time to complete the 20ms cycle

Feature Descriptions

1. HSV Color Space Recognition

The first step in color tracking is finding the target color in each frame. The project uses OpenCV's HSV (Hue-Saturation-Value) color space for color segmentation:

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv, lower_bound, upper_bound)
mask = cv2.erode(mask, None, iterations=2)    # Erosion: remove small noise
mask = cv2.dilate(mask, None, iterations=2)    # Dilation: restore target size
```

Why HSV instead of RGB? Because HSV separates a color's "hue" from its "brightness" (value) — the same red object under bright or dim light will have virtually the same H value, with only V changing. This makes color recognition more robust against lighting variations.

Special Handling for Red:

In the HSV color wheel, red straddles the H=0 boundary (approximately 170°–10°), requiring two separate masks that are then merged:

```
if hsv_lower[0] > hsv_upper[0]:    # Red crosses H=0 boundary
    mask1 = cv2.inRange(hsv, [h_low, s_low, v_low], [180, s_high, v_high])
    mask2 = cv2.inRange(hsv, [0, s_low, v_low], [h_high, s_high, v_high])
    mask = cv2.bitwise_or(mask1, mask2)
```

2. Target Localization (Minimum Enclosing Circle)

Extract contours from the color mask, select the largest contour, and compute the center of the minimum enclosing circle as the target position:

```
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
if len(contours) > 0:
    c = max(contours, key=cv2.contourArea)    # Select largest contour
    ((x, y), radius) = cv2.minEnclosingCircle(c)    # Minimum enclosing circle
    if radius > 10:    # Filter out noise contours that are too small
        # (x, y) is the target centroid
```

Why select the largest contour? Because the scene may contain multiple similarly-colored small objects or noise spots — the largest contour is typically the actual tracking target. The `radius > 10` filter ensures only targets with sufficient area are tracked.

3. PID Closed-Loop Control Algorithm

The PID (Proportional-Integral-Derivative) controller is the project's core algorithm. It takes pixel error (offset between target position and frame center) as input and outputs servo angle corrections.

Controller Structure:

```

class PIDController:
    def __init__(self, kp=0.035, ki=0.0005, kd=0.025, dead_zone=20):
        self.Kp = kp          # Proportional gain
        self.Ki = ki          # Integral gain
        self.Kd = kd          # Derivative gain
        self.dead_zone = dead_zone # Dead zone (px)
        self.integral = 0.0    # Error accumulator
        self.prev_error = 0.0  # Previous error
        self.integral_limit = 500.0 # Integral clamp (anti-windup)

```

PID Computation Process (with Hysteresis Dead Zone + Output Dead Zone):

```

def compute(self, error):
    # ① Dead zone: with hysteresis design
    if abs(error) < self.dead_zone:
        self.integral *= 0.8 # Integral decay within dead zone
        self.prev_error = error
        self._in_dead_zone = True
        return 0.0

    # ② Hysteresis band: must exceed dead_zone*1.3 to reactivate after entering dead zone
    if getattr(self, '_in_dead_zone', False) and abs(error) < self.dead_zone * 1.3:
        self.integral *= 0.9
        self.prev_error = error
        return 0.0
    self._in_dead_zone = False

    # ③ P - Proportional term
    p_out = self.Kp * error

    # ④ I - Integral term (with anti-windup clamping)
    self.integral += error
    self.integral = max(-self.integral_limit, min(self.integral_limit, self.integral))
    i_out = self.Ki * self.integral

    # ⑤ D - Derivative term
    d_out = self.Kd * (error - self.prev_error)
    self.prev_error = error

    output = p_out + i_out + d_out

    # ⑥ Output dead zone: suppress sub-degree jitter when PID output is too small
    if abs(output) < 0.3:
        return 0.0

    return output

```

Physical Meaning of the Three PID Terms:

PID Term	Mathematical Expression	Physical Effect	Parameter Impact
P (Proportional)	$K_p \cdot e(k)$	Larger error → larger correction. Provides fast response	Kp too high → overshoot oscillation; too low → sluggish response
I (Integral)	$K_i \cdot \sum e(j)$	Accumulates historical error, eliminates steady-state offset	Ki too high → integral windup oscillation; too low → residual steady-state error
D (Derivative)	$K_d \cdot [e(k) - e(k-1)]$	Senses error rate of change, suppresses overshoot	Kd too high → noise sensitive; too low → significant overshoot

Dead Zone with Hysteresis Design:

A simple dead zone (stop when error < threshold) has a serious flaw: when the target hovers around the dead zone boundary, the PID repeatedly toggles between "active-stop-active-stop", causing edge chattering. This project uses a **hysteresis dead zone** to solve this problem:

```
Error magnitude →
├─ 0 ~ 20px ────→ Inside dead zone, no action (integral ×0.8 decay)
├─ 20 ~ 26px ───→ Hysteresis band, if previously in dead zone, remain still (integral ×0.9 decay)
├─ Above 26px ───→ Activate PID, normal control
└─ Falls below 20px → Re-enter dead zone
```

Entering the dead zone requires error < 20px, but **reactivating from the dead zone requires error > 26px** (20 × 1.3). This 6px hysteresis gap eliminates the edge chattering of repeated entry and exit.

Output Dead Zone Design:

The PID computation output passes through an additional output dead zone filter: when $|u(k)| < 0.3^\circ$, it's considered "too small an angle change to act on", and outputs 0 directly. This prevents the integral term from slowly driving the servo at sub-degree jitter under minimal error.

Integral Anti-windup (Triple Protection):

The integral term is clamped within ±500. Without this limit, when the target remains at the edge of the frame for an extended time, the integral accumulates indefinitely — even when the target returns to center, the residual integral would cause the servo to "overshoot" to the opposite side — this is Integral Windup.

This implementation suppresses windup through three mechanisms:

- 1. **Hard clamping** `clamp(integral, -500, 500)` — integral never exceeds limits
- 2. **Gradual decay** `integral *= 0.8 / 0.9` — integral gradually decays within dead zone and hysteresis band
- 3. **Full reset** — when target is lost or PID parameters change, `pid.reset()` clears integral and derivative state

EMA Low-Pass Filter (Detection Coordinate Smoothing):

Before PID control, detected target coordinates are first smoothed through an **Exponential Moving Average (EMA) low-pass filter**:

```
# EMA low-pass filter parameters
ema_alpha = 0.4    # Smoothing coefficient: smaller = smoother but more latency
ema_x = None       # Initialized on first detection
ema_y = None

# Filter calculation
if ema_x is None:
    ema_x, ema_y = float(obj_x), float(obj_y) # First frame: direct assignment
else:
    ema_x = ema_alpha * obj_x + (1 - ema_alpha) * ema_x
    ema_y = ema_alpha * obj_y + (1 - ema_alpha) * ema_y
```

The EMA formula is:

$$\hat{x}(k) = \alpha \cdot x(k) + (1 - \alpha) \cdot \hat{x}(k - 1)$$

Where $\alpha = 0.4$ means the current frame's raw detection value has 40% weight, and the historical smoothed value has 60% weight.

Why is EMA needed? OpenCV's contour detection has pixel-level noise between frames — even when the target is stationary, the detected centroid may jump 2–5 pixels per frame. This high-frequency noise fed directly into PID would cause servo jitter. The EMA filter suppresses inter-frame noise while maintaining the ability to follow real target movement. $\alpha = 0.4$ is a good balance between latency and responsiveness — too small makes tracking "sluggish", too large provides insufficient noise suppression.

State Reset on Target Loss: When the target disappears from the frame, both EMA and PID states are reset (`ema_x = None`, `pid.reset()`), preventing stale historical data from producing erroneous initial control output when the target reappears.

Complete Tracking Pipeline:

```
# Pixel error (using EMA-filtered coordinates)
error_x = ema_x - center_x # Positive = target is right of center
error_y = ema_y - center_y # Positive = target is below center

# PID computation: error → angle correction
delta_pan = pid_pan.compute(error_x)
delta_tilt = pid_tilt.compute(error_y)

# Direction mapping: X right → Pan decrease (mirrored), Y down → Tilt decrease
current_pan -= delta_pan
current_tilt -= delta_tilt

# Float precision preservation + safety clamping
current_pan = clamp(current_pan, 0, 180) # Keep as float
current_tilt = clamp(current_tilt, 30, 150)

# Dedup sending: only send when angle actually changes, avoid duplicate pulse jitter
pan_int = int(current_pan)
tilt_int = int(current_tilt)
```

```

if pan_int !== prev_sent_pan:
    Bridge.call("set_pan", pan_int)
    prev_sent_pan = pan_int
if tilt_int !== prev_sent_tilt:
    Bridge.call("set_tilt", tilt_int)
    prev_sent_tilt = tilt_int

```

Key Design Details:

- Float precision preservation:** `current_pan/tilt` always maintains `float` type (initial value `90.0`), allowing PID's sub-degree corrections (e.g., 0.3°) to accumulate across frames without being truncated by int conversion. Only the final send uses `int()` conversion.
- Dedup sending:** Records the last actually sent integer angle (`prev_sent_pan/tilt`), only calling Bridge to send when the new int angle differs from the last. This avoids repeated commands when "float changed but int didn't" — repeated angle commands wouldn't change the servo position, but would cause the PWM pulse to be re-set, potentially causing micro-jitter.
- The minus sign in direction mapping:** The minus in `current_pan -= delta_pan` comes from the camera's mirror effect — a target on the right side of the frame means the camera needs to turn right, and for the Pan servo, decreasing angle corresponds to rightward rotation.

4. Web-Based Dynamic Color Selection

The Hue rainbow slider (0–180) maps to OpenCV's H channel range. When dragging the slider, ± 15 tolerance around the current Hue value defines the HSV range:

```

function hueToRange(h) {
    const hTol = 15;
    let h_low = h - hTol;
    let h_high = h + hTol;
    if (h_low < 0) h_low += 180;    // Negative wrap-around
    if (h_high > 180) h_high -= 180; // Overflow wrap-around
    return {
        h_low, s_low: 80, v_low: 80,
        h_high, s_high: 255, v_high: 255
    };
}

```

S and V lower bounds are set to 80, filtering out regions that are too gray or too dark, keeping only targets with sufficient saturation and brightness.

The 6 preset color blocks cover common tracking targets:

Preset	Hue Value	Typical Objects
Yellow	30	Tennis balls, sticky notes
Red	0	Red cups, apples
Green	60	Green markers, leaves
Blue	110	Blue folders, water bottles

Preset	Hue Value	Typical Objects
Orange	15	Oranges, safety helmets
Purple	150	Purple stationery

5. Real-Time PID Tuning Panel

The Web interface embeds a collapsible PID tuning panel with four sliders:

Parameter	Range	Step	Default	Purpose
Kp (Proportional)	0 – 0.15	0.001	0.035	Controls response speed
Ki (Integral)	0 – 0.01	0.0001	0.0005	Eliminates steady-state error
Kd (Derivative)	0 – 0.1	0.001	0.025	Suppresses overshoot oscillation
DZ (Dead Zone)	0 – 60 px	1	20	Center stillness region (with hysteresis)

Four preset configurations:

Preset	Kp	Ki	Kd	Dead Zone	Use Case
Default	0.035	0.0005	0.025	20	General balance, recommended first choice
P-only (Quick)	0.08	0	0	10	Fast response but with steady-state error and overshoot, suitable for teaching comparisons
Conservative (Smooth)	0.02	0.002	0.03	25	Smooth tracking, large dead zone, suitable for demonstrations
Aggressive (Sensitive)	0.06	0.003	0.04	10	Fast and sensitive, may overshoot, suitable for fast-moving targets

PID parameters are delivered to the Python side in real-time via Socket.IO. Switching presets automatically resets integral and derivative states to prevent jumps caused by stale state.

6. Manual PWM Servo Driving

The Arduino side does not use the Servo library, instead manually generating 50Hz PWM pulses in `loop()`:

```
void loop() {
  // Pan pulse
  digitalWrite(PIN_PAN, HIGH);
  delayMicroseconds(pulse_pan);    // HIGH duration = angle-mapped pulse width
  digitalWrite(PIN_PAN, LOW);

  // Tilt pulse
  digitalWrite(PIN_TILT, HIGH);
```

```

delayMicroseconds(pulse_tilt);
digitalWrite(PIN_TILT, LOW);

// Pad remaining time to 20ms (50Hz)
int used_time = pulse_pan + pulse_tilt;
if(20000 - used_time > 0)
    delayMicroseconds(20000 - used_time);
}

```

Angle → Pulse Width Mapping:

```

int angle_to_pulse(int angle) {
    angle = constrain(angle, 0, 180);
    return 500 + (angle * 2000 / 180);
    // 0° → 500µs, 90° → 1500µs, 180° → 2500µs
}

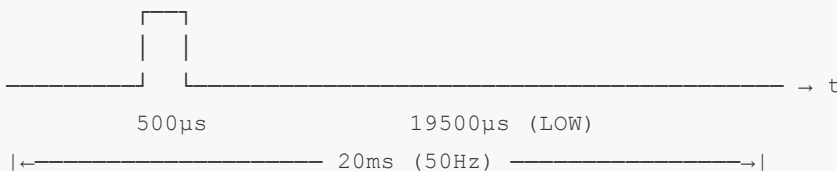
```

The MG996R servo's control protocol is compatible with the SG90: within a 20ms cycle, 0.5ms–2.5ms of HIGH pulse width linearly corresponds to 0°–180° rotation angle.

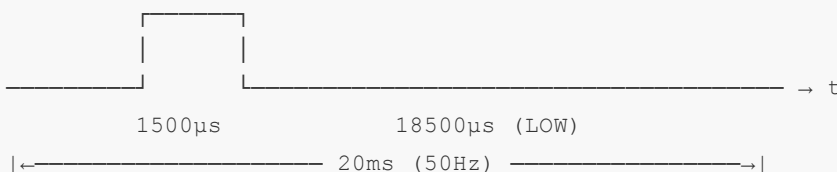
50Hz PWM Waveform Diagram:

Below are PWM waveforms for three typical angles. Each cycle is fixed at 20ms (50Hz), and the HIGH pulse width determines the servo angle:

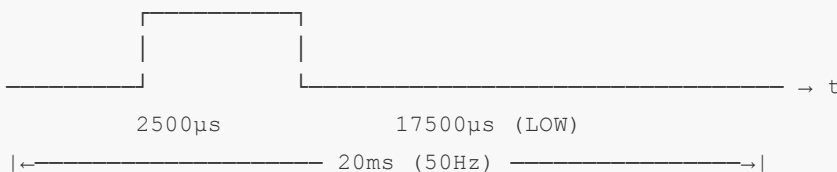
Angle = 0° (pulse width 500µs)



Angle = 90° (pulse width 1500µs)



Angle = 180° (pulse width 2500µs)



Mapping formula:

$$\text{pulse}(\theta) = 500 + \frac{\theta \times 2000}{180} \mu s, \quad \theta \in [0^\circ, 180^\circ]$$

This project outputs Pan and Tilt pulses sequentially within each 20ms cycle. The combined pulse widths are at most about 5ms, with the remaining ~15ms as LOW padding:



7. Base64 Video Streaming

Unlike the FPV surveillance robot which uses a separate video stream port, this project uses Socket.IO to directly push Base64-encoded JPEG frames:

```
ret, buffer = cv2.imencode('.jpg', frame, [int(cv2.IMWRITE_JPEG_QUALITY), 60])
if ret:
    b64_frame = base64.b64encode(buffer).decode("utf-8")
    ui.send_message('frame', {'image': b64_frame})
```

JPEG quality is set to 60 (out of 100), balancing image quality and bandwidth. Each frame is pushed at 40ms intervals (~25fps), with `time.sleep(0.04)` controlling the frame rate.

8. Camera Disconnect Tolerance Mechanism

USB cameras in embedded environments occasionally drop frames. The project implements a tolerance-based reconnection logic:

```
MAX_FAIL_COUNT = 60 # Only consider disconnected after 60 consecutive read failures

if not ret:
    fail_counter += 1
    if fail_counter > MAX_FAIL_COUNT:
        # Truly disconnected, release and re-scan
        cap.release()
        cap = None
    else:
        time.sleep(0.01) # Occasional frame drop, brief wait
        continue
else:
    fail_counter = 0 # One success, crisis averted
```

This prevents occasional frame drops from causing the camera to be frequently released and reopened, improving system stability.

Core Algorithms

This project has two core algorithms: **Two-DOF Gimbal PID Closed-Loop Control** and **HSV Color Thresholding with Centroid Computation**. Detailed mathematical principles and engineering implementation analysis follow.

Core Algorithm 1: Two-DOF Gimbal PID Closed-Loop Control

1. PID Controller Mathematical Model

The discrete position-form PID controller output formula is:

$$u(k) = K_p \cdot e(k) + K_i \cdot \sum_{j=0}^k e(j) + K_d \cdot [e(k) - e(k-1)]$$

Where:

- $e(k)$ is the pixel error at frame k (target centroid – frame center), in pixels
- $u(k)$ is the output servo angle correction, in degrees
- K_p , K_i , K_d are the proportional, integral, and derivative gains respectively

The actual system incorporates multiple protection layers:

$$u(k) = \begin{cases} 0, & |e(k)| < DZ \\ 0, & \text{in hysteresis} \wedge |e(k)| < 1.3 \times DZ \\ 0, & |PID(e(k))| < 0.3^\circ \\ PID(e(k)), & \text{otherwise} \end{cases}$$

- Line 1: **Dead zone** — no action when error is less than DZ pixels
- Line 2: **Hysteresis** — once in the dead zone, must exceed $1.3 \times DZ$ to reactivate
- Line 3: **Output dead zone** — no action when PID output is less than 0.3°
- Line 4: Normal PID control

2. EMA Low-Pass Filter — Detection Coordinate Preprocessing

Before the error enters the PID, raw detection coordinates are filtered through **Exponential Moving Average (EMA)**:

$$\hat{x}(k) = \alpha \cdot x_{\text{raw}}(k) + (1 - \alpha) \cdot \hat{x}(k-1), \quad \alpha = 0.4$$

The EMA's frequency-domain characteristic is a first-order low-pass filter with cutoff frequency approximately:

$$f_c = \frac{\alpha \cdot f_s}{2\pi(1 - \alpha)}$$

At frame rate $f_s \approx 25$ fps with $\alpha = 0.4$, $f_c \approx 2.65$ Hz — detection noise above 2.65Hz is effectively suppressed, while the typical frequency of hand-moving objects (0.5–2Hz) passes through smoothly.

Without EMA filtering, inter-frame 2–5px detection jitter would be directly amplified by PID's proportional term to $0.035 \times 5 = 0.175^\circ$ of servo tremor — while the angle is small, it's audible and visible through the MG996R's gear train.

3. Mathematical Expression of Dead Zone Hysteresis

Traditional simple dead zone:

$$u(k) = \begin{cases} 0, & |e(k)| < DZ \\ PID(e(k)), & |e(k)| \geq DZ \end{cases}$$

The problem: when the error oscillates around the DZ boundary (e.g., 19→21→19→21px), PID output switches between 0 and non-zero at high frequency, forming "edge chattering."

This project employs a **Schmitt trigger-style hysteresis dead zone**:

$$u(k) = \begin{cases} 0, & |e(k)| < DZ \\ 0, & \text{in hysteresis} \wedge |e(k)| < 1.3 \times DZ \\ PID(e(k)), & |e(k)| \geq 1.3 \times DZ \end{cases}$$

- Line 1: Enter dead zone (error < DZ)
- Line 2: Hysteresis hold (previously in dead zone, error < 1.3 × DZ)
- Line 3: Reactivate (error ≥ 1.3 × DZ)

The 6px hysteresis band between the activation threshold (1.3 × DZ = 26px) and deactivation threshold (DZ = 20px) completely eliminates edge chattering.

Integral handling within the hysteresis band is also differentiated:

- Inside dead zone: $I(k) = 0.8 \cdot I(k-1)$ (fast decay)
- Inside hysteresis band: $I(k) = 0.9 \cdot I(k-1)$ (slow decay)

The slower 0.9 decay rate prevents the integral from being cleared too quickly within the hysteresis band — if the target is about to leave again, the retained integral provides a faster initial response.

4. Integral Anti-Windup Triple Protection

An unrestricted integral term leads to the "windup" problem:

```
Target stays on right for a long time → error stays positive → integral accumulates → huge integral value
Target suddenly returns to center → error = 0 → but integral has residual → servo continues right (overshoot)
```

This implementation suppresses windup through three mechanisms:

Mechanism	Implementation	Trigger Condition
Hard clamping	<code>clamp(integral, -500, 500)</code>	After each integral accumulation
Gradual decay	<code>integral *= 0.8 / 0.9</code>	Within dead zone / hysteresis band
Full reset	<code>pid.reset()</code> → <code>integral = 0</code>	Target lost or PID parameters changed

5. Output Dead Zone — Sub-Degree Jitter Suppression

Even after hysteresis dead zone filtering, slow integral accumulation can still produce $|u(k)| < 0.3^\circ$ of tiny output. Since the servo's minimum resolution is approximately 1° (limited by PWM timing and mechanical backlash), control outputs below 0.3° have no practical physical effect and may instead produce intermittent 1° jumps due to `int()` truncation boundary effects. The output dead zone filter eliminates this phenomenon.

6. Pixel Error to Angle Correction Mapping

PID output directly represents "angle change." For the default parameter ($K_p = 0.035$):

- 100 pixel offset from center → P-term output $0.035 \times 100 = 3.5^\circ$
- 50 pixel offset from center → P-term output $0.035 \times 50 = 1.75^\circ$

- 20 pixel offset from center → P-term output $0.035 \times 20 = 0.7^\circ$ (just above output dead zone)

This proportional relationship makes tracking "big steps far away, small steps up close" — the farther the target is from center, the faster the correction; the closer, the gentler, ultimately stopping smoothly within the dead zone.

7. Float Precision Preservation and Dedup Sending

PID angle corrections are typically sub-degree (e.g., 0.35° , 0.7°). If `current_pan` used `int` type, these corrections would be truncated to 0 or 1, causing precision loss and staircase-like motion. Maintaining `float` type allows sub-degree corrections to accumulate across frames, achieving truly smooth tracking trajectories:

```
current_pan = 90.0    # float, preserving sub-degree precision
# PID output  $0.35^\circ \rightarrow 90.0 - 0.35 = 89.65 \rightarrow \text{int}() = 89$ , send 89
# Next frame PID output  $0.35^\circ \rightarrow 89.65 - 0.35 = 89.30 \rightarrow \text{int}() = 89$ , don't send (dedup)
# Next frame PID output  $0.35^\circ \rightarrow 89.30 - 0.35 = 88.95 \rightarrow \text{int}() = 88$ , send 88
```

Combined with dedup sending (`prev_sent_pan` records the last sent int value), this avoids pulse reset micro-jitter from repeated commands.

8. Physical Explanation of Direction Mapping

```
current_pan -= delta_pan    # X right (positive) → Pan angle decreases → camera turns right
current_tilt -= delta_tilt   # Y down (positive) → Tilt angle decreases → camera tilts down
```

The minus sign comes from the camera's mirror effect: a target on the right side of the frame means the camera needs to turn right, and for the Pan servo, decreasing angle corresponds to rightward rotation.

Core Algorithm 2: Color Thresholding and Centroid Computation

The visual processing for color tracking can be decomposed into five stages, forming a complete **detection pipeline**:

```
Raw BGR Frame → HSV Conversion → Color Thresholding → Morphological Denoising → Contour
Extraction → Centroid Localization
```

1. BGR to HSV Color Space Conversion

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Why HSV instead of RGB? In RGB space, the definition of "red" depends on a complex combination of three channels and is extremely sensitive to lighting — the same red object under bright light is (255,50,50), under dim light it might be (100,20,20), all three values changed. In HSV space:

Channel	Meaning	Range (OpenCV)	Characteristic
H (Hue)	Hue — the "type" of color	0–180	Nearly unchanged with lighting variations

Channel	Meaning	Range (OpenCV)	Characteristic
S (Saturation)	Saturation — the "purity" of color	0–255	Gray = 0, pure color = 255
V (Value)	Value — the "brightness" of color	0–255	Dark = 0, bright = 255

The same red object regardless of lighting conditions will always have H near 0° (i.e., OpenCV's 0), with only V changing. Therefore, only the H channel is needed for color classification, with loose lower bounds for S and V (>80) filtering out gray and overly dark areas.

2. Color Thresholding (inRange)

```
mask = cv2.inRange(hsv, lower_bound, upper_bound)
```

`inRange` performs a three-channel range check on each pixel:

$$\text{mask}(x, y) = \begin{cases} 255, & H_{\text{low}} \leq H \leq H_{\text{high}} \wedge S_{\text{low}} \leq S \leq S_{\text{high}} \wedge V_{\text{low}} \leq V \leq V_{\text{high}} \\ 0, & \text{otherwise} \end{cases}$$

The output is a binary image (white = target color pixels, black = non-target pixels).

Red Hue Wraparound Problem:

Red in the HSV color wheel straddles the H=0/180 boundary. For example, to track red (H≈0), with ±15 tolerance the range becomes H=165~15, but `inRange` doesn't support lower > upper ranges. The solution is **split + merge**:

```
if hsv_lower[0] > hsv_upper[0]: # Flag for red crossing H=0 boundary
    mask1 = cv2.inRange(hsv, [165, 80, 80], [180, 255, 255]) # H=165~180
    mask2 = cv2.inRange(hsv, [0, 80, 80], [15, 255, 255]) # H=0~15
    mask = cv2.bitwise_or(mask1, mask2) # Merge
```

The condition `hsv_lower[0] > hsv_upper[0]` is very clever — only when the Hue range crosses the 0 point (like 165~15) will the lower bound be greater than the upper bound, automatically triggering dual-segment processing. For non-wrapping colors (like yellow 20~40), lower < upper, and the normal single `inRange` is used.

3. Morphological Denoising (Opening Operation)

```
mask = cv2.erode(mask, None, iterations=2) # Erosion
mask = cv2.dilate(mask, None, iterations=2) # Dilation
```

This is the classic **opening operation** — erosion followed by dilation:

- **Erosion:** Replaces each white pixel with the minimum value in its neighborhood (3×3 default kernel). The effect is that white regions shrink, and white noise spots smaller than the kernel size are eliminated
- **Dilation:** Replaces each pixel with the neighborhood maximum. The effect is that white regions expand, restoring the target area that was shrunk by erosion

`iterations=2` means performing twice, equivalent to a single operation with a larger kernel. Two rounds of opening can eliminate random noise spots of about 4–5 pixels while preserving the integrity of large targets.

4. Contour Extraction and Maximum Area Selection

```
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
if len(contours) > 0:
    c = max(contours, key=cv2.contourArea) # Select largest contour by area
```

- `RETR_EXTERNAL`: Only extract outermost contours, ignoring nested internal contours
- `CHAIN_APPROX_SIMPLE`: Compress horizontal/vertical/diagonal segments, keeping only endpoints, reducing memory usage
- `max(key=cv2.contourArea)`: Select the largest contour by area. Multiple same-colored small objects may exist simultaneously in the scene; maximum area selection ensures the actual target is tracked rather than noise

5. Centroid Localization (Minimum Enclosing Circle Method)

```
((x, y), radius) = cv2.minEnclosingCircle(c)
if radius > 10: # Minimum area filter
    obj_x, obj_y = int(x), int(y)
```

`minEnclosingCircle` computes the **minimum radius circle** that can completely enclose the contour, returning center coordinates and radius. The `radius > 10` filter limits the minimum tracked target to approximately 10px radius (area ~314px²); smaller contours are likely residual noise.

Centroid Data Flow Summary:

```
Raw Frame (320×240) → HSV → inRange Binary Mask → Erode/Dilate Denoising
→ findContours → Max Area Contour → minEnclosingCircle
→ (obj_x, obj_y) → EMA Filtering → (ema_x, ema_y)
→ PID(ema - center) → Angle Correction → Bridge → Servo
```

The entire pipeline latency is approximately 1 frame (40ms). OpenCV operations at 320×240 resolution have negligible CPU cost.

Usage Notes and Instructions

Initial Setup:

1. Connect the two MG996R servos to Arduino UNO Q pins D3 and D11 as shown in the wiring diagram
2. Connect the USB camera to the Arduino UNO Q
3. Open Arduino App Lab and load the `color_tracked_plz` project
4. Upload the sketch firmware and Python program
5. Configure WiFi credentials

 **Important: Power-On Sequence Note**

Before powering on, if using a USB Hub expansion, pay attention to camera connection timing. It's recommended to insert the camera USB cable after the main board has fully booted.

Starting the Program:

Method 1: Via App Lab

1. Click "Run" in App Lab to start the program

Method 2: Via SSH

1. SSH into the Arduino UNO Q
2. Navigate to the app directory: `cd ArduinoApps`
3. Start the program: `arduino-app-cli app start color_tracked_plz`
4. Seeing  System Ready indicates successful startup

Accessing the Control Interface:

1. After startup, open `http://<robot-IP>:7000` in a Web browser
2. The top-right status light turning green indicates successful connection
3. The video area should display the real-time camera feed (with crosshairs)

Selecting the Tracking Color:

1. Find a brightly colored object (e.g., yellow tennis ball, red cup)
2. Drag the Hue slider in the "Tracking Color" area, or click a preset color block
3. The left color circle previews the selected color in real-time
4. The bottom-right shows the current HSV range (e.g., `H:20~40 S:80~255 V:80~255`)

Enabling Auto-Tracking:

1. Toggle on the "Auto Tracking" switch
2. Pan/Tilt sliders turn gray (under automatic control)
3. Place the target object in the camera's field of view; the gimbal tracks automatically
4. Move the object and observe the gimbal following

PID Tuning (Advanced):

1. Click "PID Tuning" to expand the panel
2. First try the 4 preset configurations to feel the effects of different parameters
3. If the default parameters don't track well, fine-tune according to these principles:
 - Response too slow? → Increase Kp
 - Jitter/oscillation? → Increase Kd, or decrease Kp
 - Target centered but with offset? → Increase Ki
 - Jitter near center? → Increase Dead Zone

Manual Mode:

1. Toggle off the "Auto Tracking" switch

2. Drag the PAN slider to control camera left-right view
3. Drag the TILT slider to control camera up-down pitch
4. Default position: PAN=90° (forward), TILT=90° (level)

Servo Center Calibration (for initial assembly):

1. Click the "Start Calibration" button → both axes auto-center to 90°
2. Auto-tracking is force-disabled, ensuring manual fine-tuning is undisturbed
3. Use $\pm 1^\circ/\pm 5^\circ$ fine-tune buttons to precisely adjust Pan and Tilt to the servo's physical center
4. Install the servo horn (cross-arm or single-arm) at the current angle
5. Click "Finish Calibration" to exit; fine-tune controls are hidden
6. To recalibrate, click  "Center 90°" for one-click reset at any time

Troubleshooting FAQ

Q: The gimbal doesn't respond after enabling auto-tracking?

A: Check if the tracking color matches the target object. Drag the Hue slider near the target object's color and ensure the object is visible in the video feed. Recognition may fail if lighting is too dim or the object's color is not saturated enough.

Q: The gimbal shakes violently during tracking?

A: PID parameters may be too aggressive. Switch to the "Conservative (Smooth)" preset, or manually decrease Kp and increase Kd and Dead Zone.

Q: "Wrong type parameter" error in the log?

A: The PID output float value was passed to Bridge without int conversion. Ensure `int()` conversion is applied before `Bridge.call("set_pan", current_pan)`.

Q: Video feed is stuttering or has high latency?

A: Base64 streaming consumes significant bandwidth at high resolutions. Ensure camera resolution is set to 320×240 and JPEG quality does not exceed 70. WiFi signal strength also affects streaming frame rate.

Q: Camera shows a blank frame?

A: Check if the camera is recognized by the system. Run `ls -l /dev/video*` via SSH — if no device nodes exist, try reinserting the USB camera.

Q: Red color tracking doesn't work well?

A: Red straddles the H=0 boundary in the HSV color wheel. The code already handles this case (dual mask merge), but large red background areas in the environment may interfere with tracking. Try narrowing the Hue tolerance or adjusting S/V lower bounds.
