Human-Centered Breeze Generator: A Technical Overview

Akhil Gharpurey, Luis Sentis, PhD, R. Babu Mummaneni, MD

November 2025

1 Introduction

This document covers the design and development of the Human-Centered Breeze Generator. This system was developed as a new means of implementing air conditioning (A/C) systems in modern homes and offices. Traditionally, residential areas offer conditioned rooms with constantly maintained temperatures. For humans, this generates physiological consequences including dehydration and allergic symptoms [1]. We sought to mitigate these issues for people in traditional working and living spaces by emulating an outdoor environment indoors. A high-level block diagram of the system is shown in Figure 1.

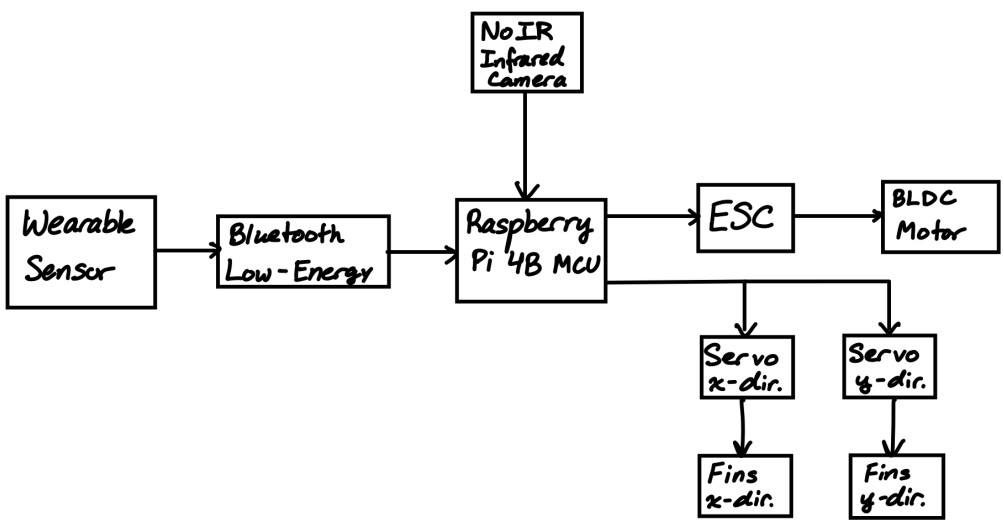

Figure 1: High-Level Block Diagram of the HCBG.

As shown in the block diagram, the HCBG is composed of three main subsystems. The master controller is a wearable sensor which senses excessive sweating through electrodermal activity (EDA) and skin temperature. The temperature and EDA data are transmitted through Bluetooth Low-Energy (BLE) to a Raspberry Pi 4B Microcontroller Unit (MCU). The Raspberry Pi acts as the central brain of the desk fan, as it parses the temperature and EDA data in real time and writes necessary PWM commands to the fan, which is actuated by a BLDC motor. Additionally, the MCU receives 2D position data of the user through real-time infrared (IR) tracking. This is used to control longitudinal fins which redirect the airflow in the direction of the user. Throughout this document, details will be provided about the design of each component which comprises the overall system.

2 Wearable Sensor

The wearable sensor is the master device in the overall system. This device was designed to measure two key thermal parameters for the user: their dermal temperature and their electrodermal activity. Electrodermal activity (EDA), also known as galvanic skin response (GSR), is a common measurement for human physiopsychological conditions based on the eccrine sweat glands [2]. These glands are actuated by the autonomic nervous system, which induces an observable electrical voltage known as the skin potential. EDA sensors measure skin potential values and interpret them as conductance values using electrodes that directly interface with the skin.

The temperature sensors used for this board are the TMP117AIDRVR sensors from Texas Instruments. These sensors feature small bounding box dimensions of 2mm x 2mm x 0.8mm and accuracy of ±0.1oC for temperature values between -20oC and 50oC, making it highly applicable for medical purposes. These sensors utilize the I2C communication protocol with customizable registers, ranging from 0x48 to 0x4B depending on the ADD0 pin connection made [3]. Because of this, the design allows for multiple temperature sensors to send data, increasing precision within the system.

The EDA sensor circuitry within the board is derived from the Grove GSR sensor, which is widely used in prototyping applications. This sensor observes a skin potential value and sends it through a feedback operational amplifier (opamp) system which transforms the observed voltages into serial analog values [4]. A 200kΩ potentiometer is also used to adjust the sensitivity of the sensor according to the user.

The wearable device must also be able to transmit physiological information to the fan wirelessly without drawing excessive current. This is done using an nRF52832 system on chip (SoC). The nRF52832 is a unique microcontroller that is specialized in wireless communication. This package features 21 general purpose input-output (GPIO) pins, 7 ADC channels, a built in low-dropout (LDO) regulator to control input voltage, and RF pins to connect to a crystal oscillator and antenna. Additionally, the SoC is programmable through Serial Wire Debug (SWD) two-wire interface [5]. We developed a real-time program using the nRF Connect Software Development Kit (SDK) which performs I2C read/write operations for temperature values and analog read operations for the user's EDA values using the SoC's built-in I2C and ADC interfaces in real time. We designed this system to send strings with the user's dermal temperature and EDA values via Bluetooth Low-Energy (BLE) at a frequency of 20 Hz to match the framerate of the Raspberry Pi detector camera. This allows for synchronization in tracked position and temperature/EDA values. If the observed skin conductance exceeds a baseline value, the nRF chip writes an output string stating "PWM ON" in addition to the temperature and EDA values. This is read by the receiver and interpreted to write the appropriate PWM signal to the motor controller.

The wearable device with the EDA sensor has a bounding box dimension of 70mm x 40mm x 1.6mm, making it fit on the average human's forearm. We additionally created an enclosure to interface this board with a wristwatch strap, creating a means of securing the wearable to the user's skin. This ensures accurate dermal temperature readings and mounting security throughout runtime.

The IR emitter is powered on the board and emits at a wavelength of 940nm and a maximum current consumption of 50 mA. We expect this to limit the battery life of the overall system to approximately 0.8 hours (48 minutes), which is problematic for long-term usage. We therefore placed a 330-ohm resistor in series with the IR LED. We found that this decreases the current load on this branch of the circuit to approximately 3.9 mA while still maintaining some brightness on the IR LED, increasing the battery life to approximately 9 hours.

3 Central Desk Fan

The central desk fan (CDF) is completely manufactured and assembled in-house using a 3D printer. The fan blades are printed using PLA plastic filament, and the fan chassis was 3D printed from ABS plastic filament. Figures 2 and 3 show a detailed CAD model of the fan assembly in isometric and rear view. The bounding box size of the CDF is 355.6mm L × 355.6mm W × 88.25mm H, and the bounding box fan blade diameter is 300mm.

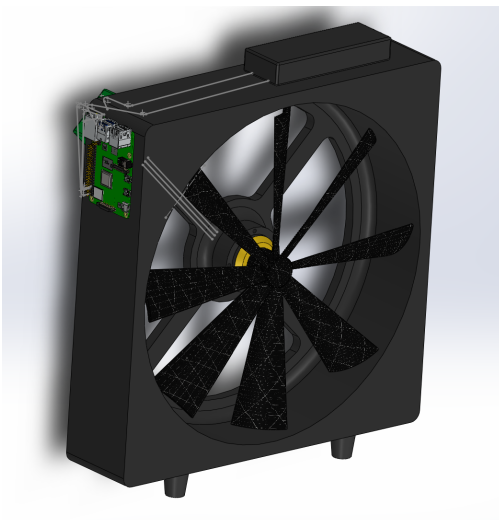

Figure 2: Isometric View of Central Desk Fan (CDF).

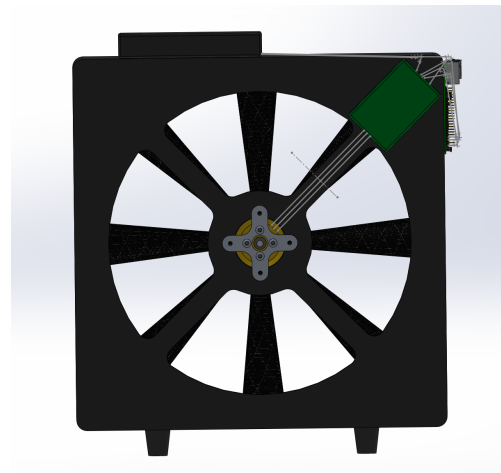

Figure 3: Rear View of Central Desk Fan (CDF).

The fan is actuated by the Scorpion-SII-4020-420KV brushless DC (BLDC) motor, which is driven by the Castle Creations Phoenix Edge 75 5V 34A Electronic Speed Controller (ESC). These actuators were selected due to their common utilization in propeller actuation in unmanned aerial vehicles (UAVs) and ability to generate high RPMs at relatively low power consumption rates. The ESC drives the motor through pulse-width modulation (PWM) with a wave frequency of 50 Hz and a duty cycle range of 5% to 10%. To test the structural integrity and efficiency in driving airflow, we simulated the fan blades in SolidWorks Flow Computational Fluid Dynamics (CFD) simulator. We used an external flow study with the assumptions of negligible flow and an angular velocity of 3,000 RPM, which roughly translates to a PWM duty cycle of 8%. Figure 4 shows the results of the CFD calculation after 222 iterations. The minimum, average, and maximum velocity for a reference plane 0.1 m from the fan and stagnation pressure values are tabulated in Table 1.

Parameter	Minimum	Average	Maximum
Velocity (m/s)	1.246	10.409	26.282
Stagnation Pressure (Pa, gage)	0.23	32.74	4007.05

Table 1: Tabulated Velocity and Pressure Values observed from CFD simulation at 3,000 RPM.

The goal of this central fan is to propel air such that a user can experience a breeze from a distance of approximately 1 meter from the fan. Typical breezes experienced at sea level range from 4 m/s to 8 m/s [6]. To verify that a breeze can be experienced by the user at the target distance, we utilized the simulation to determine the maximum distance for which the target velocity range is experienced. To achieve this goal,

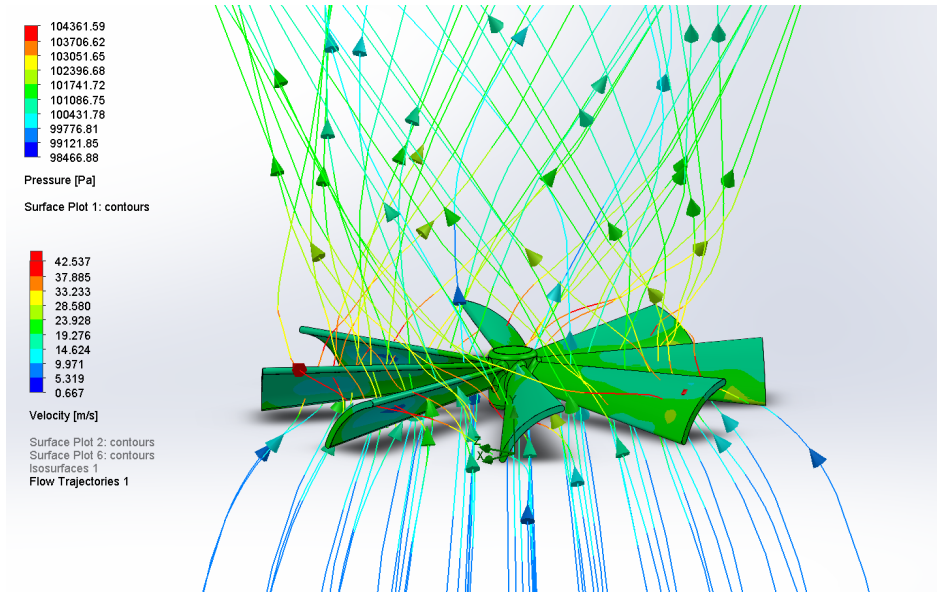

Figure 4: Velocity Trajectory and Pressure Surface Plot Results from CFD Simulation.

we constructed velocity isosurfaces at 4 m/s, 6 m/s, and 8 m/s as shown in Figure 5. Since all maximum distances are greater than 1000 mm (1.0 m), we can conclude that the fan produces a sufficient breeze for users.

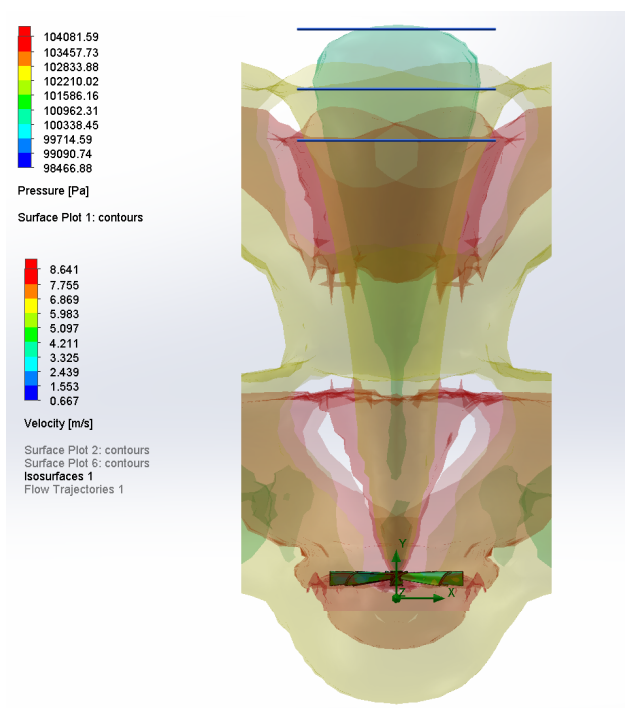

Figure 5: Isosurfaces for Breeze at a Fan Speed of 3,000 RPM. Maximum distances for velocities of 8 m/s, 6 m/s, and 4 m/s are 1.015 m, 1.085 m, and 1.275 m, respectively.

Since the fan is spinning at a high angular velocity, we must evaluate the structural integrity of each fan blade and ensure that the force exerted on them by the airflow does not cause fracture at the fillet

fixture. To achieve this, we simulated a static load case of the maximum force exerted on the fan blades. This allows us to see the maximum possible deformation and stress experienced by the fan blades. Using the CFD simulation, we obtained this force value to be 2.877 N per blade. Applying this into SolidWorks Finite Element Analysis (FEA) software, we obtain the Von Mises stress, displacement, strain, and safety factor plots shown in Figures 6, 7, 8, and 9. From these plots, we obtained a maximum Von Mises stress of 21.25 MPa and a safety factor of 1.125. Note that this is the maximum load case, meaning that our fan will survive operation at 3,000 RPM. However, we should expect the fan to fail after an infinitely long runtime, due to the presence of vibrations inducing fluctuating stresses.

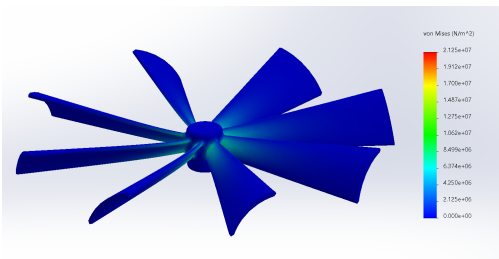

Figure 6: Von Mises Stress Plot of Fan Blades.

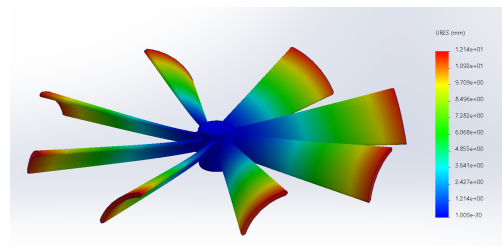

Figure 7: URES Displacement Plot of Fan Blades.

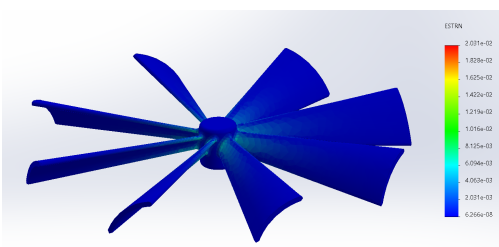

Figure 8: Equivalent Strain Plot of Fan Blades.

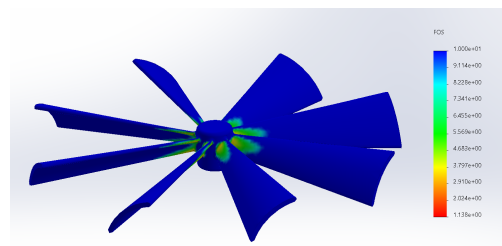

Figure 9: Factor of Safety Plot for Fan Blades.

4 Intelligent Sensor Tracker

Contemporary at-home fans utilize oscillatory mechanisms to induce convective cooling to greater regions in their indoor environment. This is often achieved through oscillation of the fan body itself or through external thin bodies which mechanically manipulate the direction of airflow, known as *fins*. In our case, the fan is meant to be maintained in a static position with a means of manipulating airflow in both the x and y directions. We therefore adopted an array of fins spanning the vertical and horizontal lengths of the central fan as modeled in Figure 10. These fins mechanically rotate about a pivot axis and drive airflow in a direction based on the user's position in real time.

4.1 Fin Rotation Mechanism

Each fin array is actuated through a DYNAMIXEL MX-64T, which drives a rack-and-pinion gear train. The rack contains 1.586 mm radius $\times$ 6 mm length pin slots symmetrically spaced at 33.02 mm intervals across the fan. The rack is constrained to a 330.2 mm L $\times$ 16.51 mm W rectangular slot, preventing rotation throughout $\mathcal{R}^3$ and only allowing for translation normal to the pin slots.

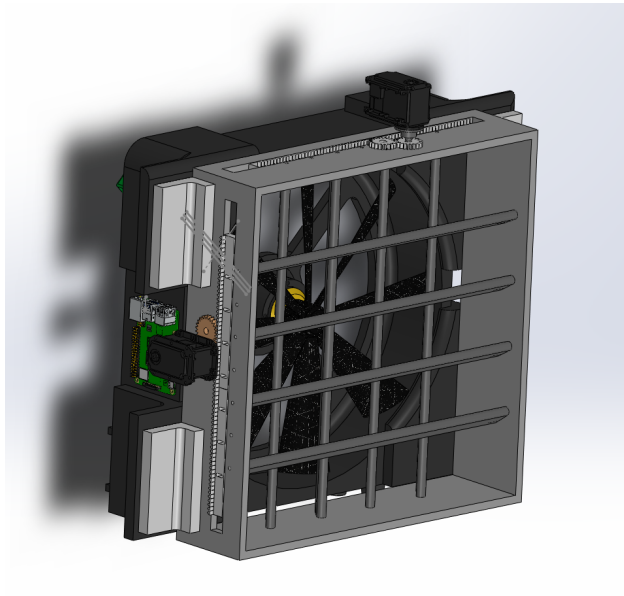

Figure 10: CAD Model of Fin Arrays Integrated with Central Fan.

To determine the maximum angular rotation for each fin array, we utilized the geometric sketch shown in Figure 11. This simplifies each fin's path motion into Cartesian motion, using $\Delta x_{\max}$ and $\Delta y_{\max}$ to denote the maximum travel in the horizontal and vertical directions. Additionally, each fin has a prescribed arc radius $R = 15$ mm, which is simply the center distance between the pivot hole and the slot hole. Additionally, the distance between the pivot hole and the center of the upper slot arc is given as 9.898 mm. The maximum vertical travel of the slotted pin will therefore be the difference of these two distances, which we find to be 5.102 mm. Using trigonometric relationships shown in Figure 11, we derive Equations 1 and 2, which allow us to calculate the maximum angular and horizontal travel of the fin using our known $\Delta y_{\max}$ value. Utilizing these equations and knowns returns a maximum angular travel $\theta_{\max} = 48.71^{\circ}$ and a maximum horizontal travel of $\Delta x_{\max} = 11.271$ mm. Therefore, this mechanism can achieve a maximum angular range of 97.42°.

$$\theta_{\max} = \cos^{-1}(1 - \frac{\Delta y_{\max}}{R}) \tag{1}$$

$$\Delta x_{\max} = R \sin\theta \tag{2}$$

The fin geometry follows a traditional airfoil design as specified according to the dimensioned drawing shown in Figure 12. We chose the airfoil geometry with a streamlined shape to minimize pressure drag that may interfere with the breeze flow and to control the lift force generated by the varying fin angular positions. When each fin rotates above or below the equilibrium position, the fin exhibits an angle of attack, α, which is used to generate lift force [7]. In our case, lift forces generate unwanted bending moments and stresses in each fin, which we must account for due to their considerable lengths. To evaluate the structural integrity of these fins, we utilized a two-step simulation process. The first step was to determine the average forcings on each face of the airfoil. We found these values through SolidWorks Flow Simulation, in which we simulated airflow of 20 m/s over the fin at an attack angle of 45° as shown in Figure 13. The resulting force values are tabulated in Table 2, in which each face number corresponds to a specific face on the airfoil length. Face 1 corresponds to the face in direct contact with the airflow, Face 2 corresponds to the larger airfoil fillet, Face 3 corresponds to the face opposite to Face 1, and Face 4 corresponds to the smaller airfoil fillet.

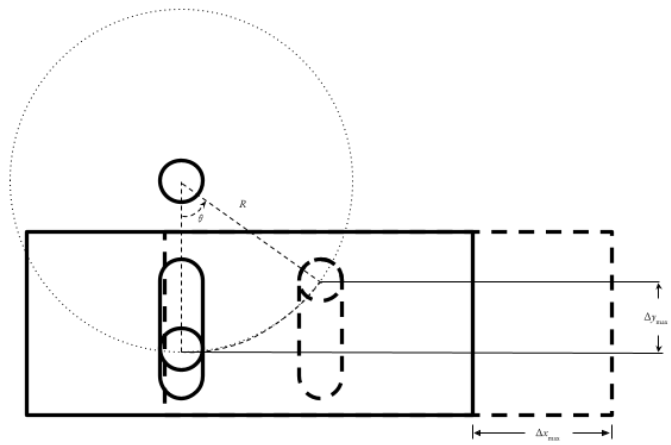

Figure 11: Geometric Sketch of Fin Rotation. The equilibrium position is given in solid lines and the maximum travel

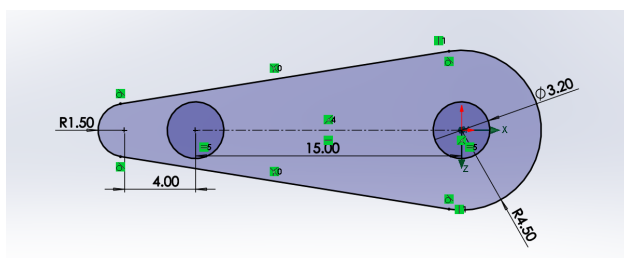

Figure 12: Cross-Sectional Geometric Sketch of Fin.

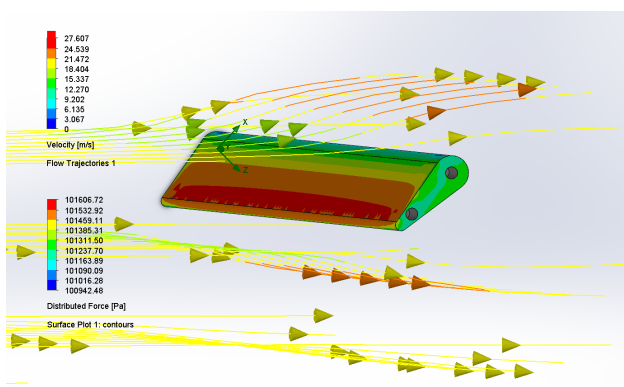

Figure 13: Stagnation Pressure Plot and Velocity Flow Trajectory over an Airfoil at an Attack Angle of 45°.

Using these forces and assuming steady-state flow conditions, we simulated these static loads using finite element analysis distributed over the fin's facial geometry. The resulting stress and safety factor plots are shown in Figures 14 and 15. The maximum Von Mises stress observed throughout the fin is found to be 104.8 kPa, and the minimum safety factor throughout the fin is approximately 160. This shows that for a breeze flow of 20 m/s, the fins can successfully withstand the airflow by a significant margin.

The fin rotation drivetrain mechanism utilizes a rack and pinion with triangular teeth geometry as

Face Number	Normal Force (N)
1	1.027
2	0.849
3	0.490
4	0.184

Table 2: Forcings on Each Fin Face due to Airflow.

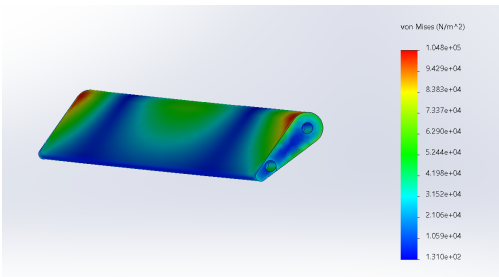

Figure 14: Von Mises Stress Plot of Fins.

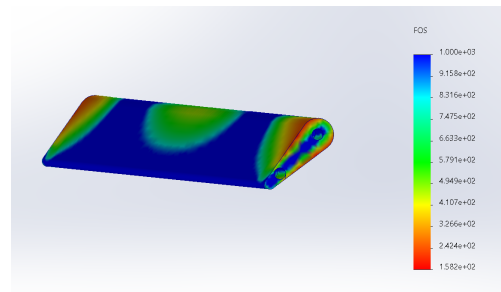

Figure 15: Safety Factor Plot of Fins.

shown in Figure 16. During actuation at a constant angular velocity, the rack will translate at a constant rate, allowing for controlled fin rotations. Table 3 denotes important parameters pertinent to each gear.

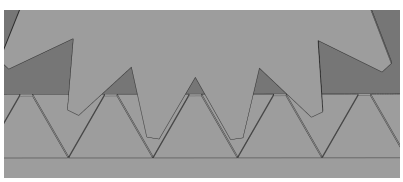

Figure 16: Rack and Pinion Tooth Profiles in Mesh.

Pitch Diameter (mm)	29.43
Number of Teeth	25
Diametral Pitch (teeth/mm)	0.849
Module (mm/teeth)	1.1772

Table 3: Relevant Gear Tooth Parameters.

4.2 IR Object Tracking Control Algorithm

The goal of this system is to force the breeze airflow in the direction of the user. In practice, the user may not be in a constant position, so we must perform this task with minimal latency and noise. We achieved this through an infrared (IR) emitter and observer pair and a real-time tracking algorithm which extracts the 2D position data of the user. The IR emitter is mounted to the wearable sensor and continuously powered during use. The IR camera then performs real-time image processing methods to determine the position of the IR emitter. These position values are taken to find the desired reference angles for which the servo motors must achieve. The full process is illustrated in the block diagram shown in Figure 17.

4.2.1 Image Processing Pipeline

Our image processing pipeline is an iterative process to extract the position coordinates of the IR emitter. The full process is illustrated in Figure 18. The initial image from the Raspberry Pi NoIR Camera is a 3D capture of the user and its surroundings. Several filters and extraction techniques are subsequently performed to extract the centroid of the best IR light blob in real time. This operation is performed at 20 Hz and matches the IR camera's set framerate of 20 frames per second (fps).

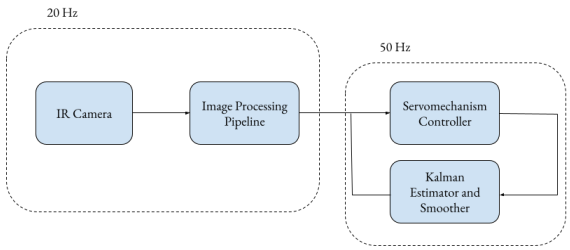

Figure 17: IR Object Tracking Algorithm Block Diagram.

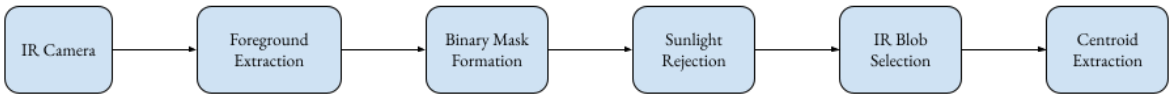

Figure 18: Image Processing Block Diagram

Initially, the Raspberry Pi NoIR Camera utilized captures an RGB image, such that each pixel is represented as a 3-dimensional vector $\mathbf{PI}_k(x, y) = [R\ G\ Bl]^T$, where R, G, and Bl represent the intensity of red, green, and blue colors between 0 and 255 for each pixel (x, y). The image is then grayscaled using the linear transformation shown in Equation 3. $I_k(x, y)$ denotes the grayscale intensity of each pixel (x, y) for some time iteration k. The image is subsequently decomposed as shown in Equation 4. $\delta_k(x, y)$ indicates any infrared (IR) disturbances found at the pixel of interest, and $n_k(x, y)$ indicates external noise captured in the image. The image's background, $B_k(x, y)$ is iteratively passed through an exponential moving-average (EMA) filter, which recursively smooths the background according to Equation 5. α is denoted as the learning rate of the EMA filter. For high values of α (e.g. $\alpha \approx 1$), the EMA filter behaves as a low-pass filter, and for low values of α, it behaves as a high-pass filter. In our case, we want to attenuate the effects of image distortion due to low-frequency drifts in the ambient environment. Therefore, we utilized a learning rate of 0.02 to ignore the low-frequency drifts in the image. The foreground is subsequently extracted by taking the difference of the original image and background, as illustrated in Equation 6.

$$I_k(x, y) = 0.299R_k(x, y) + 0.587G_x(x, y) + 0.114Bl_k(x, y) \tag{3}$$

$$I_k(x, y) = B_k(x, y) + \delta_k(x, y) + n_k(x, y) \tag{4}$$

$$B_k(x, y) = (1 - \alpha)B_{k-1}(x, y) + \alpha I_k(x, y) \tag{5}$$

$$F_k(x, y) = I_k(x, y) - B_k(x, y) \tag{6}$$

The image is subsequently passed through two decoupled filters. The original RGB image is passed through a chromatic filter, which extracts pixels which satisfy specific chromatic ratios. Since we are tracking an IR emitter, we wish to filter out all pixels that are not dominantly red in intensity. We therefore utilize the red-green and red-blue ratios specified in Equations 7 and 8 respectively. We compare these ratios for each values against critical values $\rho_g = 1.25$ and $\rho_b = 1.15$, which were tuned iteratively.

$$r_g(x,y) = \frac{R(x,y)}{G(x,y)+1} \tag{7}$$

$$r_b(x,y) = \frac{R(x,y)}{Bl(x,y)+1} \tag{8}$$

The foreground image is passed through a simple conditional luminance filter, in which each the grayscale pixel intensity, I_k, is compared to the 99th percentile of intensity within the image. To allow for fluctuations in light, a unilateral tolerance of -10 is applied, such that all pixels passed through this filter must be greater than 10 below the 99th percentile of pixel intensity in the foreground image. These filters allow us to create a binary mask as expressed in Equation 9. This mask transforms each pixel into white (full intensity) and black (zero intensity) depending on if all the specified conditions are met.

$$M_k(x,y) = (F_k(x,y) > T_f) \cap (r_g(x,y) > \rho_g) \cap (r_b(x,y) > \rho_b) \tag{9}$$

Geometries are subsequently extracted from this binary mask using mathematical morphology. In image processing, morphology is a set of operations which utilize a small structuring element E to geometrically analyze images and matrices. The basis of these operations are binary dilation and erosion, which are denoted as $A \oplus E$ and $A \ominus E$, where A is the image of interest. Binary dilation is utilized to thicken boundaries of geometric objects, resulting in smoothed contours and closed holes in the observed geometry. Binary erosion induces the opposite effect, as it shrinks the geometries' boundaries, which attenuates noise and simplifies the geometric object. In our case, we utilize morphological opening and closing of the binary mask as depicted in Equations 10 and 11, where E_o and E_c are small disk-shaped structuring elements for each respective operation [8]. Morphological openings erode and subsequently dilate the image, eliminating small geometries while maintaining the geometry of the large IR blob we are interested in observing. Morphological closing performs the dilation and erosion in the opposite order, solidifying the large IR blob region and smoothing boundaries for other present geometries. After completing these operations, we output M_f, the binary mask with the morphologically tuned geometries, including the IR blob we seek to track.

$$M_o = (M \ominus E_o) \oplus E_o \tag{10}$$

$$M_f = (M_o \oplus E_c) \ominus E_c \tag{11}$$

The resulting geometries found are decomposed into the center blob, $\mathcal{B}_i$ and the surrounding ring, $\mathcal{R}_i$. We seek to ensure that the center blob is the source of IR light emission for the object of interest, as opposed to ambient sunlight. Therefore, the mean pixel intensity of the IR blob must be greater than that of the surrounding ring. Mean pixel intensities are computed according to Equations 12 and 13. We utilize the local contrast filter depicted in Equations 14 and 15, where $\mu_{\mathcal{B}_i}$ and $\mu_{\mathcal{R}_i}$ are the mean pixel intensities of the blob and ring respectively. The constants are experimentally tuned and may be adjusted for various environments. For geometries that pass through this filter, a score $\mathcal{S}$ is computed according to Equation 16. We then extract the region with the maximum score, $\mathcal{B}^*$, using the $\arg\max$ function. This is our desired blob which we will track in real time.

$$\mu_{\mathcal{B}_i} = \frac{1}{|\mathcal{B}_i|} \sum_{(x,y)\in\mathcal{B}_i} I_k(x,y) \tag{12}$$

$$\mu_{\mathcal{R}_i} = \frac{1}{|\mathcal{R}_i|} \sum_{(x,y)\in\mathcal{R}_i} I_k(x,y) \tag{13}$$

$$\frac{\mu_{\mathcal{B}_i}}{\mu_{\mathcal{R}_i}} \geq R_\mu \tag{14}$$

$$\mu_{\mathcal{B}_i} - \mu_{\mathcal{R}_i} \geq \Delta_\mu \tag{15}$$

$$\mathcal{S}_i = |\mathcal{B}_i| \frac{\mu_{\mathcal{B}_i}}{\mu_{\mathcal{R}_i}} \tag{16}$$

The centroid of the desired region B^* is then computed according to Equation 17. As the blob is attached to the user, the expected time-series centroid values will contain significant high-frequency noise. We can model this using a measurement covariance matrix, $\mathbf{R}_k$, which is defined according to Equation 18. Here, σ_x^2 and σ_y^2 represent the measured variances of the x and y coordinates of the centroid across the blob over time. To smooth the measured output, we utilize a Kalman filter for a discrete-time linear velocity tracking model shown in Equations 19 and 20. The state vector $\mathbf{x} \equiv [x\ y\ \dot{x}\ \dot{y}]^T$, where $\dot{x}$ and $\dot{y}$ are the horizontal and vertical velocities of the IR blob respectively. Using this discrete-time model, we determine predictions for the estimated smoothed state and the uncertainty matrix for this smoothed state vector as shown in Equations 21 and 22 based on previous estimates. In this case, $\hat{\mathbf{x}}$ denotes the estimated state vector estimated from the Kalman filter, and $\mathbf{P}$ represents the state covariance matrix. We also must account for process noise, whose covariance is expressed by the matrix $\mathbf{Q}$. The Kalman filter subsequently updates these predicted values based on its innovation, which is defined as the difference between the measured output and the predicted output from the Kalman filter. The Kalman filter outputs a Kalman gain $\mathbf{L}$, which is used to create a smoothed output state vector as shown in Equations 23 and 24, which is fed as a reference input for the servomechanism controller.

$$\begin{bmatrix} \bar{x}_k \\ \bar{y}_k \end{bmatrix} = \frac{1}{|\mathcal{B}_i|} \sum_{(x,y)\in\mathcal{B}_i} \begin{bmatrix} x_k \\ y_k \end{bmatrix} \tag{17}$$

$$\mathbf{R}_k = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \tag{18}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{F}\mathbf{x}, \mathbf{F} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0.70 & 0 \\ 0 & 0 & 0 & 0.70 \end{bmatrix} \tag{19}$$

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k, \mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{20}$$

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}\hat{\mathbf{x}}_{k-1|k-1} \tag{21}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}\mathbf{P}_{k-1|k-1}\mathbf{F}^T + \mathbf{Q} \tag{22}$$

$$\mathbf{L}_k = \mathbf{P}_{k|k-1}\mathbf{H}^T(\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R})^{-1} \tag{23}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{L}_k(\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_{k|k-1}) \tag{24}$$

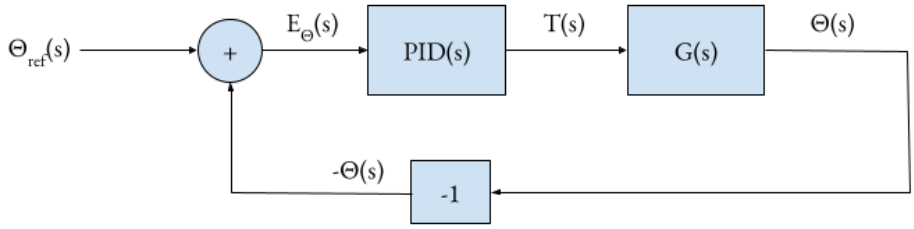

Figure 19: Feedback Controller Block Diagram for Each Servomechanism.

4.2.2 Servomechanism Feedback Controller

The servomechanism feedback controller integrates the image processing pipeline with the fin rotation mechanism. The feedback controller takes the smoothed horizontal and vertical positions of the IR emitter as a reference input and translates it into pan and tilt angles, θ_x and θ_y, which is the language of our DYNAMIXEL MX-64T 12 V Servo Motors. This is achieved by implementing a simple conversion factor shown in Equation 25. We denote W and H as the width and height of the image, in pixels. The conversion equation sets the center of the image as the origin of the 2D coordinate system, and normalizes the position data such that $(x, y) \in [-1, 1] \times [-1, 1]$, where -1 indicates the left or bottom edge, and 1 indicates the right or top edge. We utilize a conversion gain of $\pm 45°$ to achieve a wide pan and tilt range of 90° while avoiding excessive friction at the true travel boundaries found at $\theta = \pm 48.71°$.

$$\begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} = 45° \begin{bmatrix} \frac{\bar{x} - W/2}{W/2} \\ \frac{\bar{y} - H/2}{H/2} \end{bmatrix} \tag{25}$$

The feedback controller block diagram for each fin array is illustrated in Figure 19. We utilize the internal DYNAMIXEL Servo PID controller to determine the torque necessary for achieving the desired position. We model the dynamics of each servomechanism as shown in Equation 26, as derived from first principles. J is denoted as the gear shaft's mass moment of inertia, and μ is denoted as the equivalent damping coefficient. We therefore can derive the transfer function for the system dynamics as shown in Equation 27. Through SolidWorks mass properties, we find that the mass moment of inertia of the combined gear shaft and mounting disk to be approximately 0.0025 kg·m^2. We approximate the damping coefficient to be the overall coefficient of friction throughout the fin rotation mechanism. Since the rack and pinion is made out of PLA plastic and the slot is made of lubricated acrylic, we find that the total friction coefficient between all bodies is approximately 0.15. Using a 1:1 approximation, we treat the damping coefficient μ to be 0.15 N·m·s.

$$J\ddot{\theta} + \mu\dot{\theta} = \tau_{in}(t) \tag{26}$$

$$G(s) = \frac{1}{Js^2 + \mu s} \tag{27}$$

Each servo motor's PID controller has a transfer function as shown in Equation 28. The input into the PID controller is the error between the current angular position and the reference angular position, $E_\Theta(s)$, and the output is the servo motor torque, $T(s)$. The transfer function utilized implements a first-order low-pass derivative filter with filter coefficient N to prevent high-frequency oscillations passing into the torque

Gain	Rule	Value	Attenuated Value ($\beta = 2.5$)
K_p	$0.6K_u$	4.26	1.70
K_i	$\frac{1.2K_uT_s}{T_u}$	2.68	1.07
K_d	$\frac{0.075K_uT_u}{T_s}$	1.69	0.68

Table 4: PID Z-N Rules and Gains for the Servomechanism Feedback Controller.

commands. This filter coefficient is defined as the cutoff frequency of oscillation, which we found to be most effective at 6.67 Hz.

$$\text{PID}(s) = \frac{T(s)}{E_\Theta(s)} = K_p + \frac{K_i}{s} + K_d \frac{Ns}{1+Ns} \tag{28}$$

To tune the PID controller and find the appropriate gains K_p, K_i, and K_d, we implemented the Ziegler-Nichols (Z-N) method. However, to obtain accurate PID gains, we must acknowledge that the reference values are sampled at a frequency of 20 Hz, meaning that we must adapt the continuous-time transfer function into a discrete-time transfer function. Using the zero-order hold (ZOH) method, we obtain the discrete-time transfer function, $G_D(j\omega)$, as shown in Equation 29. Note that $s = j\omega$ in this case, as the Z-N method requires this equivalency to hold.

$$G_D(j\omega) = \frac{400K}{j\omega(j\omega + 60)} e^{-0.025j\omega} \tag{29}$$

To examine the system under stable oscillations, the phase lag of the system must be -180°, and the input-output gain must equal 1. Implementing these conditions allows us to solve for T_u and K_u, the ultimate period and gain of the system. These values allow us to systematically solve for the PID gains according to the Z-N rules outlined in Table 4. When applying these conditions to $G_D(j\omega)$, we ascertain ultimate gain and period values of 7.1 and 0.159 seconds. Consequently, we obtain the PID gains shown in Table 4. As these values are aggressive and induce excessive controller effort, we divide each gain by an attenuation factor of 2.5.

Using MATLAB, we simulated the stability and strength of the PID controller in following a reference trajectory. We chose a reference trajectory as modeled in Equation 30. Qualitatively, the user pans from the origin to the right side of the image in 4 seconds, and then pans to the left side of the image in 4 seconds. Figures 20 and 21 depict the system response to the tuned PID controller input over the reference signal. In Figure 21, 500 rad/s (80 Hz) noise is added to the reference signal as measurement noise. Figures 22 and 23 also show the torque output from the PID controller without and with the presence of noise.

$$\theta_{\text{ref}}(t) = \begin{cases} 10t & 0 \le t < 4 \\ 80 - 20t & 4 \le t \le 8 \end{cases} \tag{30}$$

Based on Figures 20 and 21, the PID controller stabilizes around the reference input throughout the system's simulated runtime. We observe a mean squared error of 0.115 square degrees when the system experiences noise, and a mean squared error of 0.117 square degrees without noise. Additionally, the 80 Hz noise appears to be attenuated heavily by the controller's internal low-pass filter, as the observed torque oscillation period is approximately 2.3 seconds, corresponding to an oscillation frequency of 0.43 Hz. The mean torque is approximately 1.5 N·m° during the positive slope duration and -3.0 N·m° during the negative slope duration. These are large torque magnitudes given the DYNAMIXEL MX-64T Servo's stall torque of 6.0 N·m. However,

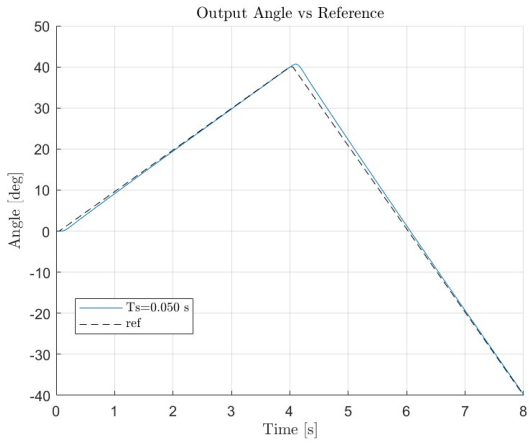

Figure 20: Controlled Path relative to Reference Path. Noise is not present.

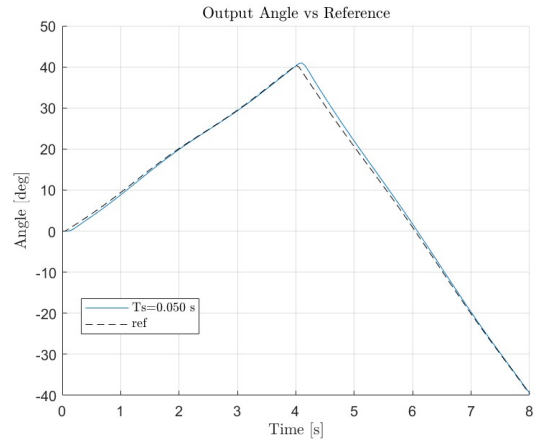

Figure 21: Controlled Path relative to Reference Path. Noise is present.

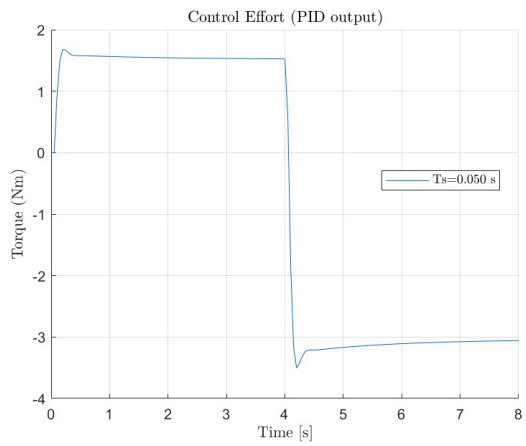

Figure 22: Controller Output for following the Reference Path. Noise is not present.

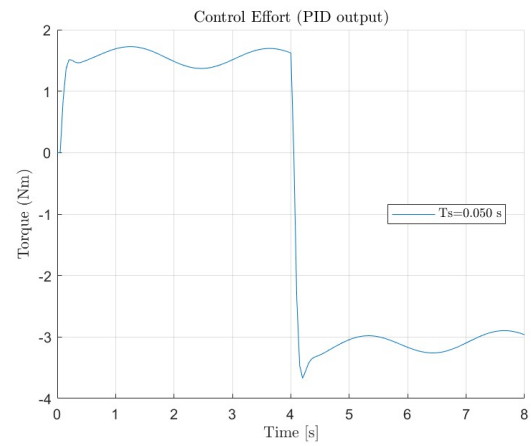

Figure 23: Controller Output for following the Reference Path. Noise is present.

it is important to note that these are based on degree measurements for the fin arrays' angular positions. If we were to convert these values to radians, we simply have to scale the plots by $\pi/180°$, or approximately 0.017 radians per degree. Performing this operation yields mean torques of 0.026 N·m and -0.052 N·m during this simulated runtime, which are far below the stall torque threshold.

5 Experimental Testing and Validation

5.1 Sensor Tracker Tests

Since there exist multiple stages in the image processing pipeline and the feedback controller, there are several sources of error we must account for and validate our model against. In the image processing pipeline, errors such as acceptance of non-desired infrared blobs or loss in tracking the desired blob. To account for these potential image processing errors, we iteratively tuned ρ_g and ρ_b, our red-green and red-blue ratio thresholds for deriving the initial binary mask to reject any infrared radiation occurring from non-red sources, such as reflected solar infrared. The infrared source we utilized is a simple resistor-diode series circuit as shown in Figure 24.From iterative testing, we found the best performance at $\rho_g = 1.35$ and $\rho_b = 1.35$. We additionally tuned the contrast ratio and difference thresholds, R_μ and Δ_μ using this iterative method. A larger R_μ and Δ_μ indicate that the central IR blob must be significantly brighter than the surrounding ring it creates. After iteratively tuning this value, we obtained that $R_\mu = 1.4$ and $\Delta_\mu = 25$.

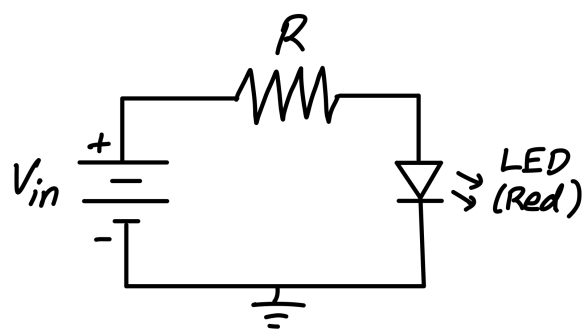

Figure 24: Schematic of Test IR Emitter. $R = 330\ \Omega,\ V_{in} = 5\ V$.

After finding these parameters, we tested the integrity of the tracking control algorithm. We conducted four 80-second independent trials in which we randomly moved the IR emitter around the camera's field of view. In trials 3 and 4, we examined the camera's ability to latch onto the emitter by turning it off and on within the trial. The figures below show the results of each trial in terms of the Kalman smoothing effect for the normalized horizontal and vertical positions and the observed servo angle position against the reference servo angle position. Figures 25-32 show the true centroidal horizontal and vertical position overlayed with the Kalman smoothed centroidal coordinates for each trial. Figures 33-40 depict the reference servo angle against the observed servo angle during operation. The deviation between these quantities is further characterized in Figures 41-44, which show the square error between the true and observed servo pan and tilt angles over time. Additionally, Table 5 shows the mean standard pan and tilt angle errors for each trial, in degrees. Note that these values are skewed due to outliers generated by losses in tracking.

Trial	Mean Standard Pan Angle Error (deg)	Mean Standard Tilt Angle Error (deg)
1	2.7083	0.9112
2	4.7734	1.1539
3	1.1008	0.7040
4	2.467	2.2447

Table 5: Mean Standard Pan and Tilt Angle Errors for Each Trial.

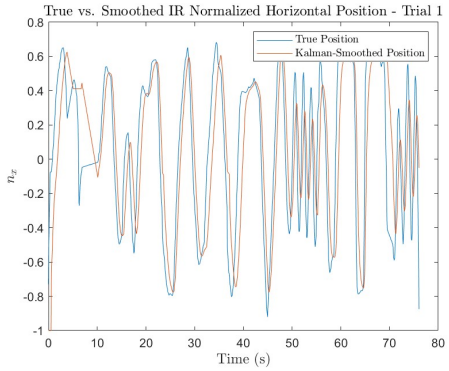

Figure 25: Normalized Horizontal Position - Trial 1

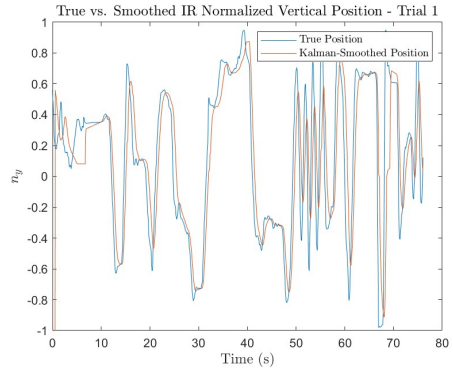

Figure 26: Normalized Vertical Position - Trial 1

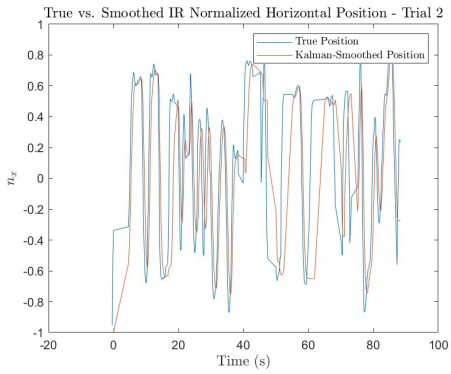

Figure 27: Normalized Horizontal Position - Trial 2

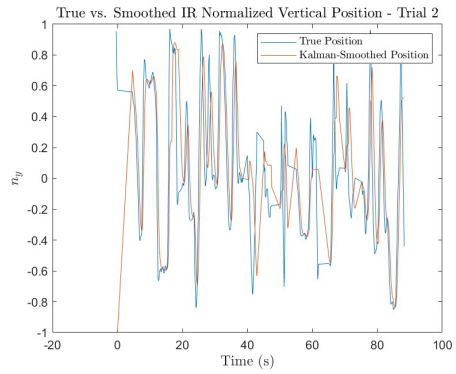

Figure 28: Normalized Vertical Position - Trial 2

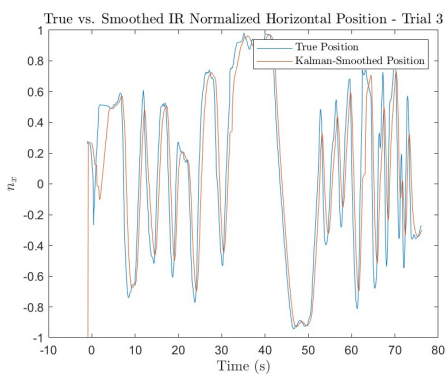

Figure 29: Normalized Horizontal Position - Trial 3

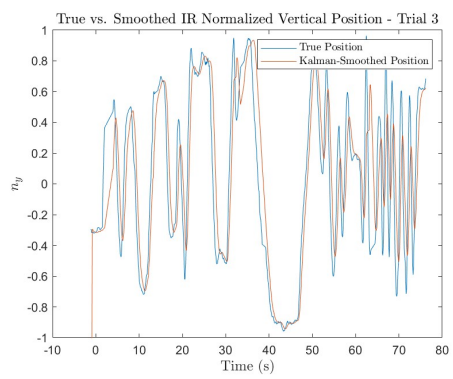

Figure 30: Normalized Vertical Position - Trial 3

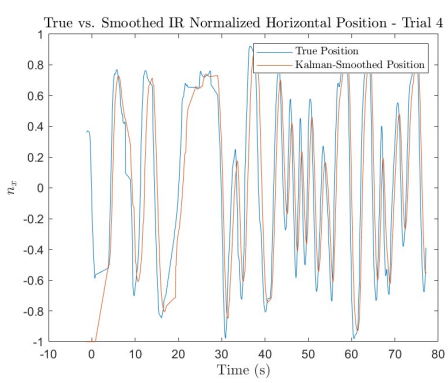

Figure 31: Normalized Horizontal Position - Trial 4

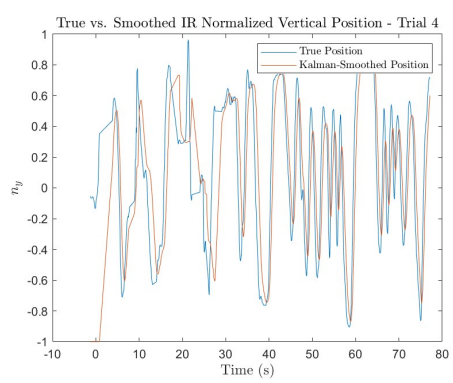

Figure 32: Normalized Vertical Position - Trial 4

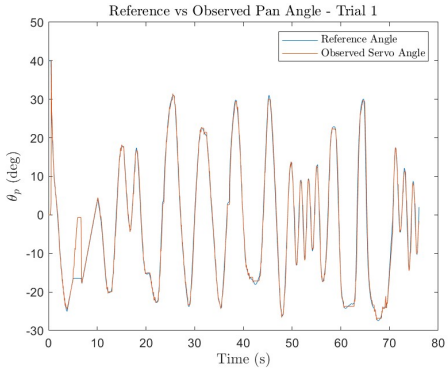

Figure 33: Reference vs. Observed Pan Angle - Trial 1

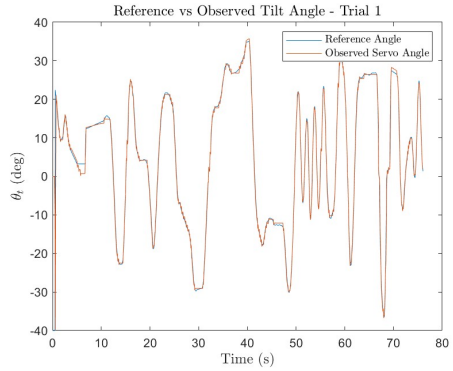

Figure 34: Reference vs. Observed Tilt Angle - Trial 1

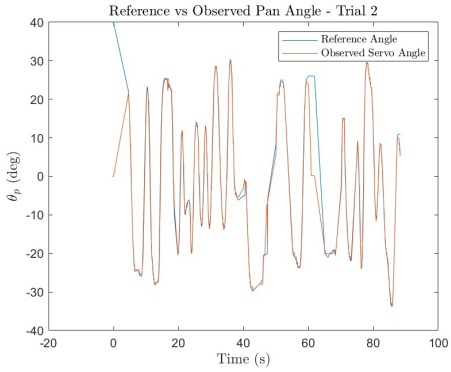

Figure 35: Reference vs. Observed Pan Angle - Trial 2

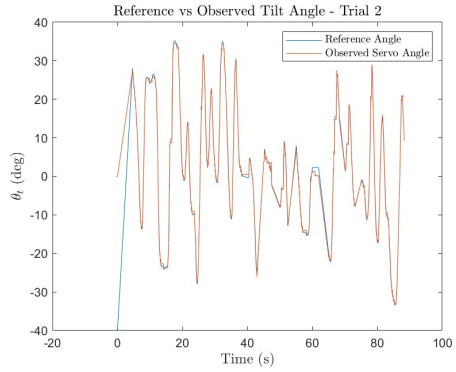

Figure 36: Reference vs. Observed Tilt Angle - Trial 2

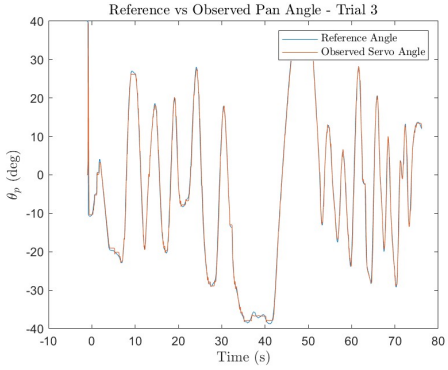

Figure 37: Reference vs. Observed Pan Angle - Trial 3

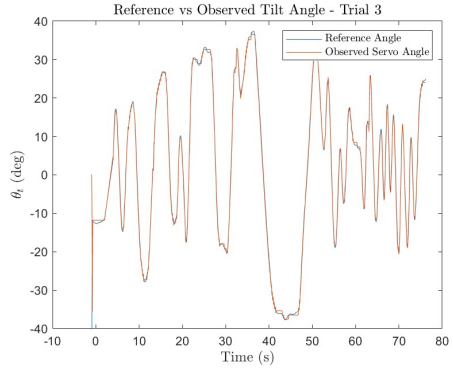

Figure 38: Reference vs. Observed Tilt Angle - Trial 3

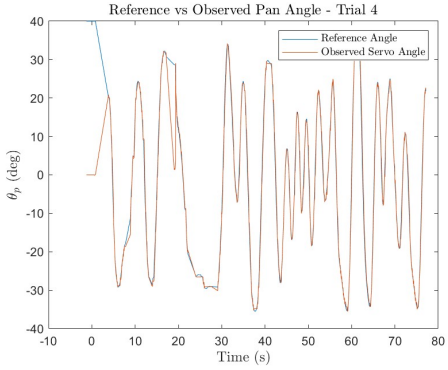

Figure 39: Reference vs. Observed Pan Angle - Trial 4

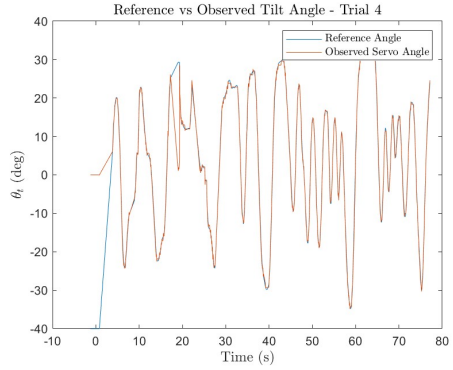

Figure 40: Reference vs. Observed Tilt Angle - Trial 4

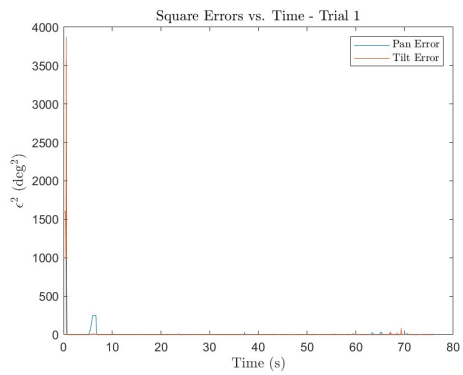

Figure 41: Pan and Tilt Angle Square Errors - Trial 1

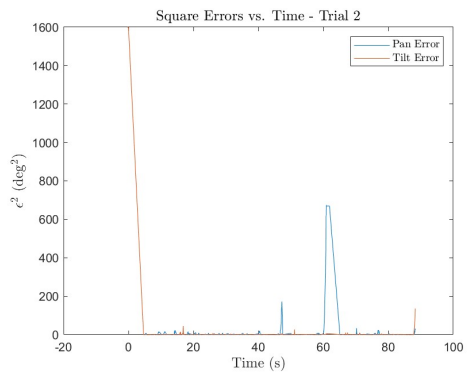

Figure 42: Pan and Tilt Angle Square Errors - Trial 2

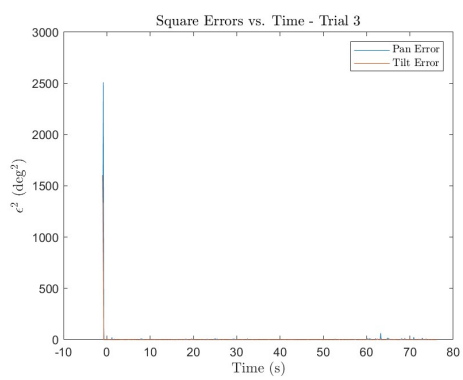

Figure 43: Pan and Tilt Angle Square Errors - Trial 3

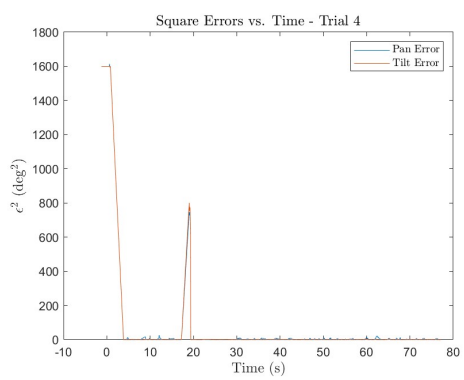

Figure 44: Pan and Tilt Angle Square Errors - Trial 4

A Sensor Component Specifications

A.1 TMP117 Specifications

Minimum Measurable Temperature ($°C$)	-20
Maximum Measurable Temperature ($°C$)	50
Temperature Sensitivity ($°C$)	± 0.1
Nominal Voltage (V)	3.3
Minimum Voltage (V)	1.8
Typical Quiescent Current (μA)	15

Table 6: Specifications for TMP117AIDRVR. [3]

A.2 Grove GSR Sensor Specifications

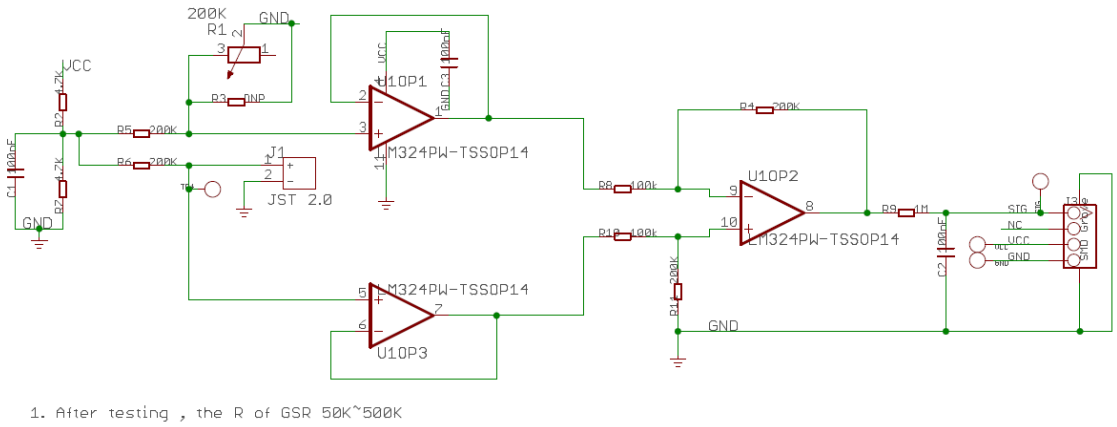

Figure 45: Schematic of Grove GSR Sensor. Nominal operating voltage is 3.3 V. [4]

A.3 nRF52832 SoC Specifications

Minimum Operating Voltage (V)	-0.3
Maximum Operating Voltage (V)	3.9
Antenna Frequency (GHz)	2.4
I2C Bitrate (kHz)	40

Table 7: Specifications of the nRF52832 SoC [5]

B Actuator Specifications

B.1 DYNAMIXEL MX-64T Specifications

Nominal Operating Voltage (V)	12.0
Minimum Operating Voltage (V)	5.0
Maximum Operating Voltage (V)	16.0
Stall Torque (N·m)	6.0
Baud Rate (bps)	57600
Initial Proportional Gain	32
Initial Integral Gain	0
Initial Integral Gain	0
Ticks per Revolution	4096
Ticks per Degree	11.4

Table 8: Specifications of DYNAMIXEL MX-64T Servo Motor [9]

.

C Software for the Human-Centered Breeze Generator

C.1 IR Tracking Software

```
# IR blob detect (sunlight rejection) + Kalman + Flask MJPEG
# + DYNAMIXEL MX-64T pan/tilt (Protocol 1.0, TTL, Tx-only writes)
# + Scanner dashboard (/dxl), Center (/dxl_center), Ensure Config (/dxl_config)

import time, threading, traceback
import numpy as np
import cv2
from flask import Flask, Response, make_response
from picamera2 import Picamera2
import csv

# ====================== DYNAMIXEL (MX-64T, Protocol 1.0) ======================
DYN_PORT = "/dev/ttyUSB0"
DYN_BAUD = 57600
PROTOCOL_VERSION  = 1.0
DX_ID_H = 1 # horizontal (pan)
DX_ID_V = 2 # vertical   (tilt)

# P1.0 control table (MX-64T)
ADDR_ID = 3
ADDR_BAUD = 4
ADDR_CW_ANGLE_LIMIT = 6
ADDR_CCW_ANGLE_LIMIT = 8
ADDR_TORQUE_ENABLE = 24
ADDR_LED = 25
ADDR_STATUS_RETURN_LVL = 16
ADDR_GOAL_POSITION     = 30
ADDR_PRESENT_POSITION  = 36
TORQUE_ENABLE = 1

ADDR_P_GAIN = 26
ADDR_I_GAIN = 27
ADDR_D_GAIN = 28

# Position scaling (0..4095  0..360°)
CENTER_TICKS  = 2048
TICKS_PER_DEG = 4096.0 / 360.0
DEG_LIMIT     = 40.0

# Command rate control
MIN_DEG_STEP  = 1.0
```

```
CMD_PERIOD_S  = 0.02   # 50 Hz

# ============================ VISION TUNABLES ============================
FRAME_SIZE = [640, 480]
FRAME_RATE = 20

# Base IR mask (intensity + chroma)
INTENSITY_PCTL   = 99.0
INTENSITY_OFFSET = -20
RG_RATIO         = 1.35
RB_RATIO         = 1.35
MIN_AREA         = max(30, int(0.0003 * FRAME_SIZE[0] * FRAME_SIZE[1]))

# Morphology
K_OPEN  = 3
K_CLOSE = 5

# Kalman
PROC_NOISE   = 0.01
MEAS_NOISE   = 3.0
MISS_DECAY_S = 0.8

JPEG_QUALITY = 80

# --------- Sunlight rejection (temporal + local contrast) ----------
LOCAL_RATIO = 1.4   # blob_mean / ring_mean must exceed this
LOCAL_DELTA = 25     # blob_mean - ring_mean must exceed this (0..60 typical)
RING_PAD    = 8      # px from blob to ring start
RING_WIDTH  = 6      # px thickness of ring

BG_ALPHA    = 0.02   # IIR background learn rate
FG_THRESH   = 20     # min (gray - background) to accept foreground

RDOM_MIN    = 0.14   # optional red-dominance floor inside blob

bg_gray = None       # lazy-init temporal background

# ============================ DYNAMIXEL SERVO COMMUNICATION BUS CLASS ============================
# Scan and find the DYNAMIXEL Servo Motors (MX-64T, Protocol 1.0, TTL)
try:
    from dynamixel_sdk import PortHandler, PacketHandler, COMM_SUCCESS
    _dx_sdk_ok = True
except Exception as e:
    print("[WARN] dynamixel-sdk not available:", e); _dx_sdk_ok = False
```

```python
# ---- utility gain conversions ----
def _clamp8(x):
    return int(max(0, min(254, round(float(x)))))

def phys_to_tbl(Kp=None, Ki=None, Kd=None):
    """
    Convert physical gains to table bytes:
      Kp  P/8,  Ki  I*(2048/1000),  Kd  D*(1000/4)
    Pass None to skip a term. Returns (P, I, D) bytes or None for skipped terms.
    """
    P = _clamp8(8*Kp)                if Kp is not None else None
    I = _clamp8(Ki * 2048.0/1000.0)  if Ki is not None else None
    D = _clamp8(Kd * 1000.0/4.0)     if Kd is not None else None
    return P, I, D

def tbl_to_phys(P=None, I=None, D=None):
    Kp = (P/8.0)          if P is not None else None
    Ki = (I*1000.0/2048)  if I is not None else None
    Kd = (D*4.0/1000.0)   if D is not None else None
    return Kp, Ki, Kd

# ============================ DYNAMIXEL MOTOR CLASS ============================
class DynBus:
    """Thread-safe, Tx-only goal writes for MX-64T (Protocol 1.0, TTL). Includes scanner."""
    def __init__(self, port=DYN_PORT, baud=DYN_BAUD, protocol=PROTOCOL_VERSION, ids=(DX_ID_H, DX_ID_V)):
        self.port_name = port
        self.baud = baud
        self.protocol = protocol
        self.ids = ids
        self.lock = threading.Lock()
        self.ok = False
        self._last_cmd = {dx_id: (None, 0.0) for dx_id in ids}
        self.last_error = ""
        self._open_port()

    # ---- Communication Port management ----
    def _open_port(self):
        if not _dx_sdk_ok:
            self.ok = False; self.last_error = "SDK not available"; return
        try:
            self.port = PortHandler(self.port_name)
            self.pkt  = PacketHandler(self.protocol)
            if not self.port.openPort():
```

```
            self.ok=False; self.last_error=f"openPort failed: {self.port_name}"; return
        if not self.port.setBaudRate(self.baud):
            self.ok=False; self.last_error=f"setBaudRate failed: {self.baud}"; return
        self.ok = True; self.last_error = ""
        print("[DXL] Port opened (P1.0, TTL).")
    except Exception as e:
        self.ok = False; self.last_error = f"Exception opening port: {e}"

def _reopen_if_needed(self):
    if self.ok: return
    print("[DXL] Reopening port...")
    try:
        self.close()
    except Exception:
        pass
    self._open_port()

# ---- Tx-only writes (no reads -> avoids rx errors) ----
def _write1_tx_only(self, dx_id, addr, val):
    self.pkt.write1ByteTxOnly(self.port, dx_id, addr, int(val))
def _write2_tx_only(self, dx_id, addr, val):
    self.pkt.write2ByteTxOnly(self.port, dx_id, addr, int(val))

# ---- SAFE READS (Protocol 1.0) for scanner/diagnostics ----
def _read1(self, dx_id, addr):
    try:
        val, res, err = self.pkt.read1ByteTxRx(self.port, dx_id, addr)
        return val, res, err
    except Exception as e:
        return None, None, e
def _read2(self, dx_id, addr):
    try:
        val, res, err = self.pkt.read2ByteTxRx(self.port, dx_id, addr)
        return val, res, err
    except Exception as e:
        return None, None, e

# Scan for servos before running commands
def scan(self, id_start=1, id_end=20):
    """Ping IDs and read torque/position. Returns list of dicts."""
    results = []
    with self.lock:
        if not self.ok:
            self._reopen_if_needed()
```

```
            if not self.ok:
                return results
        for dx_id in range(id_start, id_end + 1):
            try:
                model, fw, res, err = self.pkt.ping(self.port, dx_id)
                if res == COMM_SUCCESS and err == 0:
                    tq,_,_ = self._read1(dx_id, ADDR_TORQUE_ENABLE)
                    pos,_,_ = self._read2(dx_id, ADDR_PRESENT_POSITION)
                    results.append({
                        "id": dx_id,
                        "model": int(model),
                        "fw": int(fw),
                        "torque": (None if tq is None else int(tq)),
                        "pos": (None if pos is None else int(pos)),
                    })
            except Exception:
                pass  # ignore ping failures
    return results

# Convert degree commands to ticks comprehensible by the servo motor
def deg_to_ticks(self, deg):
    d = max(-DEG_LIMIT, min(DEG_LIMIT, float(deg)))
    return int(round(CENTER_TICKS + d * TICKS_PER_DEG))

# Allow servos to move freely (joint mode)
def ensure_joint_mode(self, dx_id):
    """SRL=1, torque off, full joint limits, torque on."""
    with self.lock:
        if not self.ok:
            self._reopen_if_needed()
            if not self.ok: return False
        try:
            try: self._write1_tx_only(dx_id, ADDR_STATUS_RETURN_LVL, 1)
            except Exception: pass
            self._write1_tx_only(dx_id, ADDR_TORQUE_ENABLE, 0)
            self._write2_tx_only(dx_id, ADDR_CW_ANGLE_LIMIT, 0)
            self._write2_tx_only(dx_id, ADDR_CCW_ANGLE_LIMIT, 4095)
            self._write1_tx_only(dx_id, ADDR_TORQUE_ENABLE, 1)
            return True
        except Exception as e:
            self.ok = False; self.last_error = f"ensure_joint_mode({dx_id}) failed: {e}"
            return False

# Write center positions for the Servos to start at
```

```python
def center(self, dx_id):
    with self.lock:
        if not self.ok:
            self._reopen_if_needed()
            if not self.ok: return
        try:
            self._write2_tx_only(dx_id, ADDR_GOAL_POSITION, CENTER_TICKS)
            self._last_cmd[dx_id] = (0.0, time.time())
        except Exception as e:
            self.ok = False; self.last_error = f"center({dx_id}) failed: {e}"

# Write degree commands to servo if significant change in observed position occurs
def maybe_send_deg(self, dx_id, deg):
    now = time.time()
    with self.lock:
        if not self.ok:
            self._reopen_if_needed()
            if not self.ok: return
        last_deg, last_t = self._last_cmd.get(dx_id, (None, 0.0))
        if last_deg is not None:
            if abs(deg - last_deg) < MIN_DEG_STEP or (now - last_t) < CMD_PERIOD_S:
                return
        ticks = self.deg_to_ticks(deg)
        try:
            self._write2_tx_only(dx_id, ADDR_GOAL_POSITION, ticks)
            self._last_cmd[dx_id] = (deg, now)
        except Exception as e:
            self.ok = False; self.last_error = f"write goal failed id={dx_id}: {e}"

# Close the communication between the Raspberry Pi controller and the servo motors
def close(self):
    with self.lock:
        try:
            for dx_id in self.ids:
                self._write1_tx_only(dx_id, ADDR_TORQUE_ENABLE, 0)
        except Exception:
            pass
        try:
            if hasattr(self, "port"):
                self.port.closePort()
        except Exception:
            pass
        self.ok = False
```

```python
# ---- Position PID (P,I,D) Control management ----
def set_position_pid(self, dx_id, *, P=None, I=None, D=None, physical=False, torque_safe=True):
    """
    Write any subset of (P,I,D) to the servo.
    Set physical=True to pass (Kp,Ki,Kd) in physical units; otherwise pass raw table bytes (0..254).
    If torque_safe=True, torque is briefly disabled during writes then restored.
    Returns (rP, rI, rD) raw bytes read back (or None if read not available).
    """
    if not self.ok:
        self._reopen_if_needed()
        if not self.ok:
            self.last_error = "Port not open";
            return (None, None, None)

    # Convert physical → table bytes if requested
    if physical:
        P, I, D = phys_to_tbl(P, I, D)

    try:
        with self.lock:
            prev_torque = None
            if torque_safe:
                # Try read torque; if failed, just attempt to disable
                val, _, _ = self._read1(dx_id, ADDR_TORQUE_ENABLE)
                prev_torque = (int(val) if val is not None else None)
                try: self._write1_tx_only(dx_id, ADDR_TORQUE_ENABLE, 0)
                except Exception: pass

            # Order on MX P1.0 table is D(26), I(27), P(28)
            if D is not None: self._write1_tx_only(dx_id, ADDR_D_GAIN, _clamp8(D))
            if I is not None: self._write1_tx_only(dx_id, ADDR_I_GAIN, _clamp8(I))
            if P is not None: self._write1_tx_only(dx_id, ADDR_P_GAIN, _clamp8(P))

            # Verify by reading back (safe read)
            rD,_,_ = self._read1(dx_id, ADDR_D_GAIN)
            rI,_,_ = self._read1(dx_id, ADDR_I_GAIN)
            rP,_,_ = self._read1(dx_id, ADDR_P_GAIN)

            if torque_safe:
                try:
                    # restore previous torque state; default to ON if unknown
                    tq = TORQUE_ENABLE if (prev_torque is None) else prev_torque
                    self._write1_tx_only(dx_id, ADDR_TORQUE_ENABLE, tq)
                except Exception:
```

```
                    pass

            return (None if rP is None else int(rP),
                    None if rI is None else int(rI),
                    None if rD is None else int(rD))
    except Exception as e:
        self.ok = False
        self.last_error = f"set_position_pid({dx_id}) failed: {e}"
        return (None, None, None)

# Read (P,I,D). If physical=True, return (Kp,Ki,Kd). These are constants which each servo uses to de
def get_position_pid(self, dx_id, *, physical=False):
    """Read (P,I,D). If physical=True, return (Kp,Ki,Kd)."""
    if not self.ok:
        self._reopen_if_needed()
        if not self.ok:
            self.last_error = "Port not open"
            return (None, None, None)
    try:
        with self.lock:
            rP,_,_ = self._read1(dx_id, ADDR_P_GAIN)
            rI,_,_ = self._read1(dx_id, ADDR_I_GAIN)
            rD,_,_ = self._read1(dx_id, ADDR_D_GAIN)
            P = (None if rP is None else int(rP))
            I = (None if rI is None else int(rI))
            D = (None if rD is None else int(rD))
            if physical:
                return tbl_to_phys(P, I, D)
            return (P, I, D)
    except Exception as e:
        self.ok = False
        self.last_error = f"get_position_pid({dx_id}) failed: {e}"
        return (None, None, None)

# Read present position and convert to degrees. Returns None on error.
def get_position_deg(self, dx_id):
    """Read present position and convert to degrees. Returns None on error."""
    if not self.ok:
        self._reopen_if_needed()
        if not self.ok:
            self.last_error = "Port not open"
            return None
    try:
        with self.lock:
```

```
                pos_ticks, res, err = self._read2(dx_id, ADDR_PRESENT_POSITION)
                if pos_ticks is None:
                    return None
                deg = (pos_ticks - CENTER_TICKS) / TICKS_PER_DEG
                return deg
        except Exception as e:
            self.ok = False
            self.last_error = f"get_position_deg({dx_id}) failed: {e}"
            return None

    def get_position_raw(self, dx_id):
        """Read present position in raw ticks (0-4095). Returns None on error."""
        if not self.ok:
            self._reopen_if_needed()
            if not self.ok:
                self.last_error = "Port not open"
                return None
        try:
            with self.lock:
                pos_ticks, res, err = self._read2(dx_id, ADDR_PRESENT_POSITION)
                return pos_ticks
        except Exception as e:
            self.ok = False
            self.last_error = f"get_position_raw({dx_id}) failed: {e}"
            return None

# ============================== KALMAN FILTER ==============================
# Creates a smoother for the IR blob position measurements to reduce jitter going into the servo motors
def make_kalman():
    kf = cv2.KalmanFilter(4, 2)
    kf.transitionMatrix  = np.array([[1,0,1,0],[0,1,0,1],[0,0,0.70,0],[0,0,0,0.70]], np.float32)
    kf.measurementMatrix = np.array([[1,0,0,0],[0,1,0,0]], np.float32)
    kf.processNoiseCov   = np.eye(4, dtype=np.float32) * PROC_NOISE
    kf.measurementNoiseCov = np.eye(2, dtype=np.float32) * MEAS_NOISE
    kf.errorCovPost      = np.eye(4, dtype=np.float32) * 1e3
    return kf

def kf_predict(kf):
    p = kf.predict(); return float(p[0]), float(p[1])

def kf_correct(kf, x, y):
    m = np.array([[np.float32(x)], [np.float32(y)]])
    kf.correct(m)
```

```python
# ========================== IR MASK (base) ============================
# Create an IR mask from BGR image using intensity + chroma criteria
def ir_mask_from_bgr(bgr):
    b, g, r = cv2.split(bgr)
    gray = cv2.cvtColor(bgr, cv2.COLOR_BGR2GRAY)

    # Intensity (auto from percentile)
    p = np.percentile(gray, INTENSITY_PCTL)
    t = np.clip(int(p + INTENSITY_OFFSET), 0, 255)
    _, mask_int = cv2.threshold(gray, t, 255, cv2.THRESH_BINARY)

    # Chroma (R dominance vs G/B)
    r_g = r.astype(np.float32) / (g.astype(np.float32) + 1.0)
    r_b = r.astype(np.float32) / (b.astype(np.float32) + 1.0)
    mask_chroma = ((r_g > RG_RATIO) & (r_b > RB_RATIO)).astype(np.uint8) * 255

    # Combine & morphology
    mask = cv2.bitwise_and(mask_int, mask_chroma)
    kopen  = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (K_OPEN,  K_OPEN))
    kclose = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (K_CLOSE, K_CLOSE))
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN,  kopen)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kclose)

    # Area prune
    cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    mask_clean = np.zeros_like(mask)
    for c in cnts:
        if cv2.contourArea(c) >= MIN_AREA:
            cv2.drawContours(mask_clean, [c], -1, 255, thickness=cv2.FILLED)
    return mask_clean

# -------------- local-contrast + foreground selection ----------------
# Check if a contour passes local contrast test according to set criteria
def passes_local_contrast(gray, contour):
    blob = np.zeros_like(gray, dtype=np.uint8)
    cv2.drawContours(blob, [contour], -1, 255, thickness=cv2.FILLED)
    outer = cv2.dilate(blob, cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
                           (RING_PAD + RING_WIDTH, RING_PAD + RING_WIDTH)))
    inner = cv2.dilate(blob, cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
                           (RING_PAD, RING_PAD)))
    ring = cv2.subtract(outer, inner)
    ring_n = int(ring.sum() // 255)
    blob_n = int(blob.sum() // 255)
    if blob_n == 0 or ring_n < 30:
```

```
        return False, (0.0, 0.0, 0.0)
    blob_mean = float(gray[blob == 255].mean())
    ring_mean = float(gray[ring == 255].mean())
    ratio = blob_mean / (ring_mean + 1e-6)
    delta = blob_mean - ring_mean
    ok = (ratio >= LOCAL_RATIO) and (delta >= LOCAL_DELTA)
    return ok, (blob_mean, ring_mean, ratio)

def passes_foreground(fg, contour):
    mask = np.zeros_like(fg, dtype=np.uint8)
    cv2.drawContours(mask, [contour], -1, 255, thickness=cv2.FILLED)
    vals = fg[mask == 255]
    if vals.size == 0:
        return False
    return (vals > FG_THRESH).mean() > 0.5

def select_best_ir_blob(mask, gray, fg, bgr):
    cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if not cnts: return None
    best, best_score = None, 0.0
    b, g, r = cv2.split(bgr)
    rdom = (r.astype(np.float32) - np.maximum(g, b).astype(np.float32)) / (r.astype(np.float32) + 1.0)
    for c in cnts:
        if cv2.contourArea(c) < MIN_AREA:
            continue
        ok_loc, stats = passes_local_contrast(gray, c)
        if not ok_loc:
            continue
        if not passes_foreground(fg, c):
            continue
        # optional red-dominance guard
        blob = np.zeros_like(gray, dtype=np.uint8)
        cv2.drawContours(blob, [c], -1, 255, thickness=cv2.FILLED)
        if rdom[blob == 255].mean() < RDOM_MIN:
            continue
        score = cv2.contourArea(c) * stats[2]  # area × local ratio
        if score > best_score:
            best_score, best = score, c
    if best is None:
        return None
    M = cv2.moments(best)
    if M["m00"] == 0:
        return None
    cx = M["m10"]/M["m00"]; cy = M["m01"]/M["m00"]
```

```
    x,y,w,h = cv2.boundingRect(best)
    return (cx, cy), (x, y, w, h), best

# ============================== CAMERA ==============================
# Initialize Raspberry Pi IR Camera
picam2 = Picamera2()
cfg = picam2.create_video_configuration(
    main={"size": FRAME_SIZE},
    controls={"FrameRate": FRAME_RATE},
    buffer_count=4
)
picam2.configure(cfg)
picam2.start()
time.sleep(0.5)
picam2.set_controls({
    "AeEnable": False,
    "AwbEnable": False,
    "ColourGains": (1.4, 1.4),
    "Saturation": 0.4,
    "ExposureTime": 10000,
    "AnalogueGain": 1.0
})

# Global data storage for CSV export
_csv_data = {
    'runtime': [],
    'true_x_position': [],
    'true_y_position': [],
    'filtered_x_position': [],
    'filtered_y_position': [],
    'servo_angle_x': [],
    'servo_angle_y': [],
    'filename': 'ir_tracker_data.csv',
    'lock': threading.Lock()
}

def write_csv_data():
    """Write accumulated data to CSV file."""
    with _csv_data['lock']:
        if len(_csv_data['runtime']) == 0:
            return
        try:
            with open(_csv_data['filename'], "a", newline="") as f:
                writer = csv.writer(f)
```

```
                for t, tx, ty, fx, fy, ax, ay in zip(
                    _csv_data['runtime'],
                    _csv_data['true_x_position'],
                    _csv_data['true_y_position'],
                    _csv_data['filtered_x_position'],
                    _csv_data['filtered_y_position'],
                    _csv_data['servo_angle_x'],
                    _csv_data['servo_angle_y']
                ):
                    writer.writerow([t, tx, ty, fx, fy, ax, ay])
            # Clear arrays after writing
            _csv_data['runtime'].clear()
            _csv_data['true_x_position'].clear()
            _csv_data['true_y_position'].clear()
            _csv_data['filtered_x_position'].clear()
            _csv_data['filtered_y_position'].clear()
            _csv_data['servo_angle_x'].clear()
            _csv_data['servo_angle_y'].clear()
            print(f"[CSV] Data written to {_csv_data['filename']}")
        except Exception as e:
            print(f"[ERROR] Could not write CSV: {e}")

# ============================ PIXEL → ANGLES ============================
# Convert pixel coordinates to servo angles
def pixel_to_angles(px, py, W, H):
    nx = (px - (W/2.0)) / (W/2.0)   # [-1..+1]
    ny = ((H/2.0) - py) / (H/2.0)   # flip Y so up is +1
    theta_x = max(-DEG_LIMIT, min(DEG_LIMIT, -40.0 * nx))
    theta_y = max(-DEG_LIMIT, min(DEG_LIMIT, -25.0 * ny - 15.0))
    return theta_x, theta_y

# ================================ APP ================================
# Main Flask app + global state
# Initialize all objects and variables used in application

app = Flask(__name__)
kf = make_kalman()
last_meas_time = 0.0
bus = DynBus()
atten = 2 # servo PID attenuation factor

# Write PID gains (physical units) to servos
bus.set_position_pid(DX_ID_H, P=4.33/atten, I=2.75/atten, D=1.7/atten, physical=True, torque_safe=True)
bus.set_position_pid(DX_ID_V, P=4.33/atten, I=2.75/atten, D=1.7/atten, physical=True, torque_safe=True)
```

```
# Configure & center once
for dxid in (DX_ID_H, DX_ID_V):
    bus.ensure_joint_mode(dxid)
    bus.center(dxid)

# ============================ MJPEG GENERATOR ============================
# Create live video MJPEG stream with IR blob detection and servo control
def mjpeg_generator():
    global last_meas_time, bg_gray
    last_hb = 0.0
    last_csv_write = 0.0

    # Initialize CSV file with headers
    try:
        with open(_csv_data['filename'], "w", newline="") as f:
            writer = csv.writer(f)
            writer.writerow(["Timestamp", "TrueX", "TrueY", "FilteredX", "FilteredY", "AngleX", "AngleY'
        print(f"[CSV] Initialized {_csv_data['filename']}")
    except Exception as e:
        print(f"[ERROR] Could not initialize CSV: {e}")

    while True:
        try:
            t_now = time.time()
            _csv_data['runtime'].append(t_now)
            # lightweight bus heartbeat
            if time.time() - last_hb > 2.0 and not bus.ok:
                bus._reopen_if_needed()
                last_hb = time.time()

            # Periodic CSV write (every 5 seconds or 100 records)
            if len(_csv_data['runtime']) > 0 and (time.time() - last_csv_write > 5.0 or len(_csv_data['r
                write_csv_data()
                last_csv_write = time.time()

            rgb = cv2.flip(picam2.capture_array(),0)
            bgr = cv2.cvtColor(rgb, cv2.COLOR_RGB2BGR)
            H, W = bgr.shape[:2]

            # --- temporal background & foreground ---
            gray = cv2.cvtColor(bgr, cv2.COLOR_BGR2GRAY)

            early_mask = cv2.inRange(gray, 230, 255)
            present_early = np.count_nonzero(early_mask) > 15
```

```
if bg_gray is None:
    bg_gray = gray.astype(np.float32)
else:
    if not present_early:
        cv2.accumulateWeighted(gray, bg_gray, BG_ALPHA)
fg = cv2.subtract(gray, bg_gray.astype(np.uint8))

# Base mask, then strong selection with sunlight rejection
mask = ir_mask_from_bgr(bgr)
meas = select_best_ir_blob(mask, gray, fg, bgr)

# Kalman
px, py = kf_predict(kf)
px = max(0, min(W-1, px))
py = max(0, min(H-1, py))
if meas is not None:
    (cx, cy), (x, y, w, h), _ = meas
    kf_correct(kf, cx, cy)
    _csv_data['true_x_position'].append(cx)
    _csv_data['true_y_position'].append(cy)
    last_meas_time = time.time()
    cv2.rectangle(bgr, (x, y), (x+w, y+h), (0,255,0), 2)
    cv2.circle(bgr, (int(cx), int(cy)), 4, (0,0,255), -1)
    cv2.putText(bgr, f"IR {w}x{h}px", (x, max(20, y-8)),
                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255,255,255), 2)
else:
    if time.time() - last_meas_time > MISS_DECAY_S:
        cv2.putText(bgr, "PREDICTING (no measurement)", (12, 28),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0,255,255), 2)

# Draw predicted centroid
cv2.circle(bgr, (int(px), int(py)), 4, (0,255,0), -1)
_csv_data['filtered_x_position'].append(px)
_csv_data['filtered_y_position'].append(py)

# Map to servos
theta_x, theta_y = pixel_to_angles(px, py, W, H)

dt_miss = time.time() - last_meas_time
if dt_miss < 0.5:
    # normal tracking
    if bus.ok:
        bus.maybe_send_deg(DX_ID_H, theta_x)
```

```
            bus.maybe_send_deg(DX_ID_V, theta_y)
        else:
            # blob missing → slow down to avoid drifting away
            scale = max(0.0, 1.0 - (dt_miss - 0.5) / 1.0)   # fade over 1s
            if scale > 0:
                if bus.ok:
                    bus.maybe_send_deg(DX_ID_H, theta_x * scale)
                    bus.maybe_send_deg(DX_ID_V, theta_y * scale)

        # Get actual servo positions
        h_pos = bus.get_position_deg(DX_ID_H)
        v_pos = bus.get_position_deg(DX_ID_V)

        if h_pos is not None and v_pos is not None:
            _csv_data['servo_angle_x'].append(h_pos)
            _csv_data['servo_angle_y'].append(v_pos)
        else:
            # Append None or 0 if read fails to keep arrays aligned
            _csv_data['servo_angle_x'].append(0.0 if h_pos is None else h_pos)
            _csv_data['servo_angle_y'].append(0.0 if v_pos is None else v_pos)

        # Small mask inset (blue channel)
        small = cv2.resize(mask, (0,0), fx=0.4, fy=0.4, interpolation=cv2.INTER_NEAREST)
        sh, sw = small.shape
        y0, x0 = 10, 10
        roi = bgr[y0:y0+sh, x0:x0+sw]
        if roi.shape[:2] == small.shape:
            roi[:,:,0] = 0; roi[:,:,1] = 0; roi[:,:,2] = small

        # HUD
        cv2.putText(bgr, f"PAN:{theta_x:+.1f}°  TILT:{theta_y:+.1f}°", (12, 54),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255,255,255), 2)

        ok, jpg = cv2.imencode(".jpg", bgr, [int(cv2.IMWRITE_JPEG_QUALITY), JPEG_QUALITY])
        if not ok:
            continue
        yield (b"--frame\r\nContent-Type: image/jpeg\r\n\r\n" + jpg.tobytes() + b"\r\n")

    except GeneratorExit:
        break
    except Exception:
        traceback.print_exc()
        continue
```

```
# ============================ FLASK ROUTES ============================
# Main Streaming Page
@app.route("/")
def index():
    html = """<!doctype html>
<title>IR Tracker + DYNAMIXEL (Sunlight Rejection)</title>
<style>
  html,body{background:#000;height:100%;margin:0;display:grid;place-items:center}
  img{max-width:100%;height:auto}
</style>
<img src="/video" alt="stream"/>
<p style="position:fixed;bottom:12px;left:12px;font:14px/1.2 system-ui;color:#ddd">
<a href="/dxl" style="color:#8ff;text-decoration:none;border:1px solid #355;padding:6px 10px;border-radi
</p>
"""
    resp = make_response(html)
    resp.headers["Cache-Control"] = "no-store, no-cache, must-revalidate, max-age=0"
    return resp

@app.route("/video")
def video():
    return Response(
        mjpeg_generator(),
        mimetype="multipart/x-mixed-replace; boundary=frame",
        headers={"Cache-Control":"no-store, no-cache, must-revalidate, max-age=0",
                 "Pragma":"no-cache"}
    )

# -------------------- DXL Scanner & Quick Actions --------------------
@app.route("/dxl")
def dxl():
    ID_START, ID_END = 1, 20   # widen if needed
    results = bus.scan(ID_START, ID_END)
    rows = []
    for r in results:
        rows.append(
            f"<tr><td>{r['id']}</td>"
            f"<td>{r['model']}</td>"
            f"<td>{r['fw']}</td>"
            f"<td>{r['torque']}</td>"
            f"<td>{r['pos']}</td></tr>"
        )
    html = f"""<!doctype html>
<title>DYNAMIXEL Scanner</title>
```

```
<style>
body{{font-family:system-ui,Segoe UI,Arial;padding:16px;background:#111;color:#eee}}
table{{border-collapse:collapse;width:100%}}
td,th{{border:1px solid #333;padding:8px}}
thead th{{background:#222}}
.ok{{color:#8f8}} .err{{color:#f88}}
a.btn{{display:inline-block;padding:6px 10px;border:1px solid #444;border-radius:8px;text-decoration:nor
</style>
<h1>DYNAMIXEL Scanner</h1>
<p>Port: <b>{bus.port_name}</b> @ {DYN_BAUD} | Status: <b class="{ 'ok' if bus.ok else 'err' }">{'OK' if
<p>
  <a class="btn" href="/dxl">Refresh</a>
  <a class="btn" href="/dxl_center">Center (IDs {DX_ID_H}, {DX_ID_V})</a>
  <a class="btn" href="/dxl_config">Ensure Config (IDs {DX_ID_H}, {DX_ID_V})</a>
  <a class="btn" href="/">Back to Stream</a>
</p>
<table>
  <thead><tr><th>ID</th><th>Model</th><th>FW</th><th>Torque</th><th>PresentPos</th></tr></thead>
  <tbody>{''.join(rows) if rows else f'<tr><td colspan=5>No servos in {ID_START}..{ID_END}</td></tr>'}</
</table>
<p>Tip: If nothing appears but Wizard works, check Protocol=1.0, baud={DYN_BAUD}, TTL port, and Status P
"""
    resp = make_response(html)
    resp.headers["Cache-Control"] = "no-store, no-cache, must-revalidate, max-age=0"
    return resp

@app.route("/dxl_center")
def dxl_center():
    for dxid in (DX_ID_H, DX_ID_V):
        bus.center(dxid)
    return make_response('<meta http-equiv="refresh" content="0; url=/dxl">', 200)

@app.route("/dxl_config")
def dxl_config():
    for dxid in (DX_ID_H, DX_ID_V):
        bus.ensure_joint_mode(dxid)
    return make_response('<meta http-equiv="refresh" content="0; url=/dxl">', 200)

@app.route("/health")
def health(): return "ok", 200

if __name__ == "__main__":
    try:
        app.run(host="0.0.0.0", port=5000, threaded=True)
```

```
finally:
    try:
        bus.close()
    except Exception:
        pass
    try:
        cv2.destroyAllWindows()
        picam2.stop()
        # Write any remaining data
        if len(_csv_data['runtime']) > 0:
            write_csv_data()
            print(f"[CSV] Final write complete")
    except Exception:
        pass
```

C.2 Raspberry Pi BLE Communication Software

```python
#!/usr/bin/env python3
"""
BLE temperature receiver → ESC PWM (GPIO 13 via pigpio):
- Scans & reconnects to BLE_Alert forever
- Parses Avg=xx.xx from notifications
- temp > 30.0°C -> throttle 25% (1.25ms pulse); else throttle 0% (1.0ms)
"""

import asyncio, re, signal, sys, time
import pigpio
from bleak import BleakScanner, BleakClient

# Nordic UART (NUS)
NUS_SERVICE_UUID = "6e400001-b5a3-f393-e0a9-e50e24dcca9e"
NUS_TX_CHAR_UUID = "6e400003-b5a3-f393-e0a9-e50e24dcca9e"  # notify

# PWM / ESC settings
PWM_PIN       = 13         # BCM
MIN_PULSE_US  = 1000       # 1.00 ms
MAX_PULSE_US  = 2000       # 2.00 ms
ARM_TIME_S    = 1.5        # hold 1.00 ms at startup

pi = None
running = True

def set_servo_pct(pct: float):
    """Set ESC throttle 0..1 -> 1.0..2.0 ms pulse at 50 Hz via pigpio."""
    pct   = max(0.0, min(1.0, pct))
    pulse = int(MIN_PULSE_US + (MAX_PULSE_US - MIN_PULSE_US) * pct)
    pi.set_servo_pulsewidth(PWM_PIN, pulse)
    print(f"[PWM] throttle={pct:.2f} -> {pulse} us")

def arm_esc():
    """Basic arming: send 1.00 ms for a moment."""
    print("[PWM] Arming ESC at 1.00 ms ...")
    set_servo_pct(0.0)
    time.sleep(ARM_TIME_S)

def parse_and_act(line: str):
    """Extract Avg=xx.xx and set throttle based on threshold."""
    m = re.search(r"Avg\s*=\s*([+-]?\d+(?:\.\d+)?)", line)
    if not m:
        return
```

```
    temp_c = float(m.group(1))
    pct = 0.25 if temp_c > 30.0 else 0.0
    set_servo_pct(pct)
    print(f"[TEMP] {temp_c:.2f} °C -> throttle {pct*100:.0f}%")

def sig_handler(signum, frame):
    global running
    running = False

async def run_loop():
    # pigpio daemon must be running: sudo systemctl enable --now pigpiod
    global pi
    pi = pigpio.pi()
    if not pi.connected:
        print("pigpio daemon not running. Start with: sudo systemctl enable --now pigpiod")
        sys.exit(1)
    pi.set_mode(PWM_PIN, pigpio.OUTPUT)
    arm_esc()

    while running:
        print("Scanning for BLE_Alert...")
        devices = await BleakScanner.discover()
        target = next((d for d in devices if d.name and "BLE_Alert" in d.name), None)
        if not target:
            print("BLE_Alert not found. Retrying in 3s...")
            await asyncio.sleep(3)
            continue

        print(f"Connecting to {target.name} ({target.address})...")
        try:
            async with BleakClient(target.address) as client:
                await client.start_notify(NUS_TX_CHAR_UUID,
                    lambda _, data: parse_and_act(data.decode('utf-8', errors='replace')))
                print("Subscribed to notifications. Ctrl+C to quit.")
                # keep alive until interrupted or exception
                while running and client.is_connected:
                    await asyncio.sleep(1)
        except Exception as e:
            print(f"[BLE] Error: {e}. Reconnecting in 3s...")
            await asyncio.sleep(3)

    # On exit, send safe pulse (optional cut to 0)
    set_servo_pct(0.0)
    pi.stop()
```

```
    print("[PWM] Stopped.")

def main():
    signal.signal(signal.SIGINT,  sig_handler)
    signal.signal(signal.SIGTERM, sig_handler)
    asyncio.run(run_loop())

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

C.3 Wearable Sensor Firmware

C.3.1 main.c

```
#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/devicetree.h>
#include <zephyr/drivers/i2c.h>
#include <zephyr/drivers/adc.h>
#include <zephyr/bluetooth/bluetooth.h>
#include <zephyr/bluetooth/hci.h>
#include <zephyr/bluetooth/conn.h>
#include <zephyr/bluetooth/gatt.h>
#include <zephyr/bluetooth/services/nus.h>
#include <zephyr/logging/log.h>
#include <string.h>
#include <stdio.h>

/* ========= Logging ========= */
LOG_MODULE_REGISTER(tmp117_app, LOG_LEVEL_INF);

/* ========= I2C / TMP117 ========= */
#define I2C_NODE         DT_NODELABEL(i2c0)

#define TMP117_ADDR1     0x48   /* ADD0 = GND */
#define TMP117_ADDR2     0x49   /* ADD0 = VCC */

#define TMP117_REG_TEMP      0x00
#define TMP117_REG_CONFIG    0x01
#define TMP117_REG_DEVICE_ID 0x0F

#define TMP117_DEVICE_ID_EXPECTED 0x0117
#define TMP117_LSB_C        0.0078125f   /* 7.8125 m°C per LSB */

/* Threshold in °C for PWM_ON in the message */
#define THRESHOLD_C         26.0f

static const struct device *i2c_dev;

/* ========= ADC (SAADC) via devicetree =========
 * overlay defines:
 *
 * / {
 *   zephyr_user: user {
 *     compatible = "zephyr,user";
```

```
 *     io-channels = <&adc 5>;
 *   };
 * };
 *
 * AIN5 on nRF52832 DK corresponds to P0.07.
 */
static const struct adc_dt_spec adc_chan = ADC_DT_SPEC_GET(DT_PATH(zephyr_user));
static int16_t adc_raw_buf;
static struct adc_sequence adc_seq;

/* ========= BLE (NUS) ========= */
static struct bt_conn *current_conn;

/* NUS callback: data received from central */
static void nus_rx_cb(struct bt_conn *conn, const void *data,
                      uint16_t len, void *ctx)
{
    ARG_UNUSED(conn);
    ARG_UNUSED(ctx);

    /* Just log it for now */
    char buf[64];

    uint16_t copy_len = (len < sizeof(buf) - 1) ? len : (sizeof(buf) - 1);
    memcpy(buf, data, copy_len);
    buf[copy_len] = '\0';

    LOG_INF("NUS RX: %s", buf);
}

/* NUS callback: central subscribed/unsubscribed to notifications */
static void nus_notif_enabled(bool enabled, void *ctx)
{
    ARG_UNUSED(ctx);
    LOG_INF("NUS notifications %s", enabled ? "ENABLED" : "DISABLED");
}

/* Zephyr NUS callbacks (Zephyr 3.x / NCS 3.1 style) :contentReference[oaicite:0]{index=0} */
static struct bt_nus_cb nus_cb = {
    .notif_enabled = nus_notif_enabled,
    .received      = nus_rx_cb,
};

/* Connection callbacks */
```

```c
static void connected(struct bt_conn *conn, uint8_t err)
{
    if (err) {
        LOG_ERR("BLE connect failed (err %u)", err);
        return;
    }

    current_conn = bt_conn_ref(conn);
    LOG_INF("BLE connected");
}

static void disconnected(struct bt_conn *conn, uint8_t reason)
{
    ARG_UNUSED(reason);

    LOG_INF("BLE disconnected (reason 0x%02x)", reason);
    if (current_conn) {
        bt_conn_unref(current_conn);
        current_conn = NULL;
    }
}

BT_CONN_CB_DEFINE(conn_cbs) = {
    .connected    = connected,
    .disconnected = disconnected,
};

/* ========= BLE advertising setup ========= */

static const struct bt_data ad[] = {
    BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
};

static const struct bt_data sd[] = {
    BT_DATA(BT_DATA_NAME_COMPLETE,
            CONFIG_BT_DEVICE_NAME,
            sizeof(CONFIG_BT_DEVICE_NAME) - 1),
};

static void bt_ready(int err)
{
    if (err) {
        LOG_ERR("Bluetooth init failed (err %d)", err);
        return;
```

```
    }

    LOG_INF("Bluetooth ready");

    err = bt_nus_cb_register(&nus_cb, NULL);
    if (err) {
        LOG_ERR("bt_nus_cb_register failed (err %d)", err);
        return;
    }

    err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad),
                          sd, ARRAY_SIZE(sd));
    if (err) {
        LOG_ERR("Advertising failed to start (err %d)", err);
    } else {
        LOG_INF("Advertising as '%s'", CONFIG_BT_DEVICE_NAME);
    }
}

/* ========= TMP117 helpers ========= */

static int tmp117_read_reg16(uint8_t addr, uint8_t reg, uint16_t *out)
{
    uint8_t buf[2];
    int ret = i2c_write_read(i2c_dev, addr, &reg, 1, buf, 2);
    if (ret) {
        return ret;
    }

    *out = ((uint16_t)buf[0] << 8) | buf[1];
    return 0;
}

static int tmp117_write_reg16(uint8_t addr, uint8_t reg, uint16_t val)
{
    uint8_t buf[3];

    buf[0] = reg;
    buf[1] = (uint8_t)((val >> 8) & 0xFF);
    buf[2] = (uint8_t)(val & 0xFF);

    return i2c_write(i2c_dev, buf, sizeof(buf), addr);
}
```

```c
/* Optional: configure TMP117 for continuous conversion, etc. */
static int tmp117_init(uint8_t addr)
{
    uint16_t id;
    int ret = tmp117_read_reg16(addr, TMP117_REG_DEVICE_ID, &id);
    if (ret) {
        LOG_ERR("TMP117(0x%02X) ID read failed (err %d)", addr, ret);
        return ret;
    }

    if (id != TMP117_DEVICE_ID_EXPECTED) {
        LOG_WRN("TMP117(0x%02X) unexpected ID=0x%04X", addr, id);
    } else {
        LOG_INF("TMP117 detected at 0x%02X", addr);
    }

    /* You can also write TMP117_REG_CONFIG here if you want non-default mode.
     * For many boards, the default-config continuous mode is fine.
     */

    return 0;
}

/* Read temperature in °C from one TMP117 */
static int tmp117_read_c(uint8_t addr, float *temp_c)
{
    uint16_t raw16;
    int ret = tmp117_read_reg16(addr, TMP117_REG_TEMP, &raw16);
    if (ret) {
        return ret;
    }

    /* TMP117 is a signed 16-bit value in 2's complement. */
    int16_t raw = (int16_t)raw16;
    *temp_c = (float)raw * TMP117_LSB_C;
    return 0;
}

/* ========= ADC helpers (SAADC) ========= */

static int adc_init(void)
{
    int err;
```

```
    if (!adc_is_ready_dt(&adc_chan)) {
        LOG_ERR("ADC device not ready");
        return -ENODEV;
    }

    err = adc_channel_setup_dt(&adc_chan);
    if (err) {
        LOG_ERR("adc_channel_setup_dt failed (err %d)", err);
        return err;
    }

    /* Initialize the sequence for this channel */
    err = adc_sequence_init_dt(&adc_chan, &adc_seq);
    if (err) {
        LOG_ERR("adc_sequence_init_dt failed (err %d)", err);
        return err;
    }

    adc_seq.buffer = &adc_raw_buf;
    adc_seq.buffer_size = sizeof(adc_raw_buf);

    LOG_INF("ADC initialized (chan=%u, res=%u bits)",
            adc_chan.channel_id, adc_chan.resolution);

    return 0;
}

/* Read single sample in millivolts */
static int adc_read_mv(int32_t *mv_out)
{
    int err;

    err = adc_read_dt(&adc_chan, &adc_seq);
    if (err) {
        LOG_ERR("adc_read_dt failed (err %d)", err);
        return err;
    }

    int32_t mv = adc_raw_buf;

    err = adc_raw_to_millivolts_dt(&adc_chan, &mv);
    if (err) {
        LOG_ERR("adc_raw_to_millivolts_dt failed (err %d)", err);
        return err;
```

```
    }

    *mv_out = mv;
    return 0;
}

/* ========= Main ========= */

int main(void)
{
    int err;

    LOG_INF("2x TMP117 + ADC + BLE_Alert (NUS)");

    /* I2C device */
    i2c_dev = DEVICE_DT_GET(I2C_NODE);
    if (!device_is_ready(i2c_dev)) {
        LOG_ERR("I2C device not ready");
        return 0;
    }

    /* Explicitly set 100 kHz (can also be done in devicetree) */
    (void)i2c_configure(i2c_dev,
                        I2C_MODE_CONTROLLER |
                        I2C_SPEED_SET(I2C_SPEED_STANDARD));

    /* Init TMP117 sensors */
    (void)tmp117_init(TMP117_ADDR1);
    (void)tmp117_init(TMP117_ADDR2);

    /* Init ADC */
    (void)adc_init();

    /* Init BLE */
    err = bt_enable(bt_ready);
    if (err) {
        LOG_ERR("bt_enable failed (err %d)", err);
    }

    /* Let sensors power up */
    k_sleep(K_MSEC(10));

    while (1) {
        float t1_c = 0.0f;
```

```c
float t2_c = 0.0f;
int r1 = tmp117_read_c(TMP117_ADDR1, &t1_c);
int r2 = tmp117_read_c(TMP117_ADDR2, &t2_c);

bool have1 = (r1 == 0);
bool have2 = (r2 == 0);

if (!have1) {
    LOG_WRN("TMP117 @0x48 read failed (%d)", r1);
}
if (!have2) {
    LOG_WRN("TMP117 @0x49 read failed (%d)", r2);
}

float avg_c = 0.0f;
if (have1 && have2) {
    avg_c = 0.5f * (t1_c + t2_c);
} else if (have1) {
    avg_c = t1_c;
} else if (have2) {
    avg_c = t2_c;
} else {
    avg_c = 0.0f;
}

bool alert = (avg_c > THRESHOLD_C);

int32_t adc_mv = 0;
int adc_ret = adc_read_mv(&adc_mv);
if (adc_ret) {
    LOG_WRN("ADC read failed (%d)", adc_ret);
}

/* Build message for NUS + RTT log.
 * Example: "T1=24.92 C T2=25.10 C Avg=25.01 C ADC=1234 mV PWM_OFF"
 */
char msg[80];

snprintf(msg, sizeof(msg),
         "T1=%.2f C T2=%.2f C Avg=%.2f C ADC=%" PRId32 " mV %s",
         t1_c, t2_c, avg_c, adc_mv,
         alert ? "PWM_ON" : "PWM_OFF");

LOG_INF("%s", msg);
```

```
        /* Send over BLE if there is a connection and notifications are enabled.
         * bt_nus_send() will return -ENOTCONN if nobody subscribed. :contentReference[oaicite:1]{index=
         */
        if (current_conn) {
            int tx_err = bt_nus_send(current_conn, msg, strlen(msg));
            if (tx_err && tx_err != -ENOTCONN) {
                LOG_WRN("bt_nus_send err %d", tx_err);
            }
        }

        k_sleep(K_SECONDS(1));
    }

    return 0;
}
```

C.3.2 customboard.overlay and customboard.yaml (for devicetree configuration)

```
/dts-v1/;
#include <nordic/nrf52832_qfaa.dtsi>

/ {
    model = "Custom nRF52832 Board";
    compatible = "customboard";

    chosen {
        zephyr,console = &uart0;
        zephyr,shell-uart = &uart0;
    };

    /* User ADC node for easy SAADC access */
    zephyr_user: user {
        compatible = "zephyr,user";
        io-channels = <&adc 5>;  /* AIN5 = P0.07 */
    };
};

/* ================================
 * CLOCK CONFIGURATION (CRITICAL)
 * ================================ */
&clock {
    /* No 32k crystal → use LFRC */
    lfclk {
        lfclk-source = "LFRC";
    };

    /* You DO have a 32 MHz crystal → enable HFXO */
    hfclk {
        hfclk-source = "HFXO";
    };
};

/* ================================
 * Enable UART0 console
 * ================================ */
&uart0 {
    status = "okay";
    current-speed = <115200>;
};

/* ================================
```

```
 * I2C0 on P0.23 (SDA) & P0.22 (SCL)
 * ================================ */
&pinctrl {
    i2c0_default: i2c0_default {
        group1 {
            psels = <
                NRF_PSEL(TWIM_SDA, 0, 23)
                NRF_PSEL(TWIM_SCL, 0, 22)
            >;
        };
    };

    i2c0_sleep: i2c0_sleep {
        group1 {
            psels = <
                NRF_PSEL(TWIM_SDA, 0xff, 0xff)
                NRF_PSEL(TWIM_SCL, 0xff, 0xff)
            >;
        };
    };
};

&i2c0 {
    status = "okay";
    pinctrl-0 = <&i2c0_default>;
    pinctrl-1 = <&i2c0_sleep>;
    pinctrl-names = "default", "sleep";
    clock-frequency = <I2C_BITRATE_STANDARD>;
};

/* ADC peripheral enabled */
&adc {
    status = "okay";
};


identifier: customboard
name: "Custom nRF52832 Board"
type: mcu
arch: arm
ram: 64
flash: 512
toolchain:
  - zephyr
  - gnuarmemb
```

```
  - xtools
supported:
  - ble
  - pwm
  - i2c
  - adc
```

C.3.3 prj.conf (configuration parameters)

```
# ========== Core ==========
CONFIG_MAIN_STACK_SIZE=2048
CONFIG_HEAP_MEM_POOL_SIZE=2048

# ========== I2C ==========
CONFIG_I2C=y

# ========== ADC / SAADC ==========
CONFIG_ADC=y
CONFIG_ADC_NRFX_SAADC=y

# ========== Bluetooth / NUS ==========
CONFIG_BT=y
CONFIG_BT_PERIPHERAL=y
CONFIG_BT_DEVICE_NAME="BLE_Alert"
CONFIG_BT_MAX_CONN=1

# Nordic UART Service
CONFIG_BT_ZEPHYR_NUS=y

# Keep security simple while prototyping
CONFIG_BT_SMP=n
CONFIG_BT_SETTINGS=n

# Allow some payload room for your temperature string
CONFIG_BT_L2CAP_TX_MTU=100

# ========== Logging / console ==========
CONFIG_LOG=y
CONFIG_LOG_DEFAULT_LEVEL=3
CONFIG_PRINTK=y
CONFIG_CONSOLE=y

# Use RTT as log backend (optional but nice on Nordic DKs)
CONFIG_USE_SEGGER_RTT=y
CONFIG_RTT_CONSOLE=y
CONFIG_UART_CONSOLE=n
```

References

[1] V. Ganji, M. Kalpana, U. Madhusudhan, N. A. John, and M. Taranikanti, "Impact of air conditioners on sick building syndrome, sickness absenteeism, and lung functions.," *PubMed*, vol. 27, pp. 26–30, Jun. 2023. DOI: 10.4103/ijoem.ijoem_23_22. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10257240/#:~:text=Our%20study%20showed%20that%20the.

[2] Y. S. Can, B. Arnrich, and C. Ersoy, "Stress detection in daily life scenarios using smart phones and wearable sensors: A survey," *Journal of Biomedical Informatics*, vol. 92, p. 103 139, Apr. 2019. DOI: 10.1016/j.jbi.2019.103139. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1532046419300577.

[3] T. Instruments, *Tmp117 high-accuracy, low-power, digital temperature sensor with smbus™-and i 2 c-compatible interface*. Accessed: Nov. 8, 2025. [Online]. Available: https://www.ti.com/lit/ds/symlink/tmp117.pdf?ts=1762488879450.

[4] S. Studio, *Grove -gsr sensor*, 2013. [Online]. Available: https://files.seeedstudio.com/wiki/Grove-GSR_Sensor/res/Grove-GSR_Sensor_WiKi.pdf.

[5] N. Semiconductor, *Technical documentation*, Nordicsemi.com, 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/nRF52832_PS_v1.9/resource/nRF52832_PS_v1.9.pdf.

[6] F. Barrow, *Sea breeze circulation*, RMetS, Jun. 2024. [Online]. Available: https://www.rmets.org/metmatters/sea-breeze-circulation.

[7] T. Liu, S. Wang, H. Liu, and G. He, "Engineering perspective on bird flight: Scaling, geometry, kinematics and aerodynamics," *Progress in Aerospace Sciences*, vol. 142, p. 100 933, Oct. 2023. DOI: 10.1016/j.paerosci.2023.100933. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0376042123000490?casa_token=pv4Cj_SY9k4AAAAA:B2no-qzZmipaUUfxbW5Z7QPOqUvlvA6yXmpYWEUZdM3VMBG90N

[8] R. Haralick, "Mathematical morphology and computer vision," *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp. 468–479, 2025. DOI: 10.1109/acssc.1988.754039. Accessed: Nov. 8, 2025. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=754039.

[9] ROBOTIS, *Robotis e-manual*, ROBOTIS e-Manual. [Online]. Available: https://emanual.robotis.com/docs/en/dxl/mx/mx-64/.