Motion Control for Visual Tracking

Visual Recording Object Oriented Mapping

BSc Thesis

D. Al-Rushdy
F. Nezamie

TU Delft

Motion Control for Visual Tracking

BSc Thesis

by

Dany Al-Rushdy & Ferdaws Nezamie

Student Name	Student Number
Dany Al-Rushdy	5870836
Ferdaws Nezamie	5876788

Supervisors:
Dr. Padmakumar R. Rao
Dr. Sandra K. Raveendran
Tejus Kusur

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science

Monday 16th June, 2025

Abstract

This thesis presents the design and implementation of a motion control system intended for a real-time vision-based tracking application. The goal was to track a coloured, free-falling water droplet using low-cost hardware. For initial system verification, a coloured splash ball was used as a proxy target, chosen for its higher visibility and consistent shape. The system integrated a Raspberry Pi 5 with a Raspberry Pi HQ Camera and a Pimoroni PIM183 pan–tilt unit for vertical target tracking. A PD controller adjusted the tilt angle based on positional data to track and centre the target. To compensate for approximately 100 ms of actuation latency, along with additional delays introduced by processing and computation, the PD controller used predicted target positions. These were provided by an Extended Kalman Filter, which was configured to forecast motion 160 ms ahead. Experimental results showed that the splash ball remained in view for approximately 40–50% of its fall duration. Accurate centring was not achieved, as delays in actuator response limited the system's ability to keep pace with the high velocity of the target. Furthermore, water droplet tracking proved to be infeasible, as the detection system could not detect such small targets. These findings indicate that, due to hardware-induced delays, the system was unable to achieve stable tracking of high-velocity targets.

Preface

This thesis is written as part of the Bachelor Graduation Project of the Electrical Engineering programme at Delft University of Technology. The project focuses on combining embedded control systems, real-time computer vision and predictive tracking methods.

The idea for choosing this project came from an interest in how low-cost hardware, like the Raspberry Pi, can be used in high-performance tracking systems. During the project, we had the opportunity to engage in both the theory behind control design and the practical side of building and integrating the system.

We would like to express our gratitude to our supervisors Dr. Padmakumar R. Rao, Dr. Sandra K. Raveendran and Tejus Kusur for their guidance and support during this project. We would also like to thank the members of the collaborating subgroup: Enes Kaya, Naufal El Khatibi and Timo Haas for a pleasant collaboration throughout the quarter.

Contents

1 Introduction

Real-time object tracking using active camera systems is a fundamental challenge in computer vision and control, with applications in areas such as video surveillance and unmanned aerial vehicles (UAVs) [1], [2]. The specific task of tracking a free-falling object introduces significant constraints on both detection accuracy and control speed due to rapid acceleration under gravity and the limited time window in which the object remains visible. To keep a rapidly exiting free-falling object in view, active cameras require extremely fast and precise control.

Visual servoing, a control methodology where visual feedback is used to guide camera orientation, has proven effective in keeping moving targets within the camera frame [3]. Dynamic camera systems outperform static setups by increasing tracking accuracy and extending the operational workspace [4]. However, high-speed visual tracking systems often rely on specialized hardware platforms such as FPGAs and galvanometer mirrors to meet stringent real-time requirements [5], [6]. While these platforms are effective, they are high in costs.

1.1. State-of-the-Art

In embedded systems, classical control strategies such as PID controllers are frequently combined with predictive estimation techniques like Kalman filters. These methods have been successfully applied to track moderately fast objects using pan–tilt platforms in real-time scenarios [7], [8]. However, their performance are rarely evaluated under more demanding conditions, such as those involving gravitational acceleration and strict actuator latency constraints.

Recent research has explored more advanced predictive control strategies to address the limitations of traditional feedback methods. Nebeluk et al. present a model predictive control (MPC) approach for pan–tilt camera systems, employing second-order linear models to forecast target motion over a finite horizon. Their method maintains the object near the image centre by tightly integrating visual feedback with actuator prediction and demonstrates improved tracking performance in both simulation and physical robot experiments [9].

When tracking small and fast-moving objects, such as water droplets, additional challenges arise due to motion blur and low signal-to-noise ratios. Mirzaei et al. provide a comprehensive review of these issues and emphasize that classical detection methods often fail under such conditions. They advocate for predictive approaches, including Kalman and particle filters, which are particularly effective when operating with low-resolution visual input and under mechanical latency constraints [10].

1.2. Project Objectives and Challenges

The objective of this project was to design and implement a real-time tracking system capable of maintaining a free-falling object centred within the frame of a camera throughout its fall. This was initially tested with a coloured ball and was intended for tracking a water droplet. The motivation for tracking a falling water droplet stems from its relevance to research in fluid dynamics and atmospheric sciences, where detailed observation of droplet behaviour is vital for validating physical models related to atomization, rainfall and air–fluid interactions [11].

The tracking system was designed to run on embedded hardware, utilizing low-cost components. This platform was chosen to evaluate the feasibility of high-speed visual tracking using accessible hardware. However, the use of an embedded platform imposes strict limitations on system performance. The selected servo mechanism introduce an inherent actuation delay of up to 20 ms, which constrains the control loop's responsiveness. When combined with delays from image acquisition and processing, this further degrades real-time performance. Additionally, objects falling under gravity reach high velocities, demanding fast and precise control updates. Together, these constraints pose significant challenges to achieving accurate and responsive real-time tracking.

1.3. Thesis Structure

This thesis is organized as follows. Chapter 2 defines the Programme of Requirements, detailing both functional constraints and performance metrics that guided the development. Chapter 3 outlines the overall system design and methodology, including a description of the hardware configuration. Chapter 4 provides a detailed analysis of the actuator used for motion tracking. Chapter 5 discusses the selection of an appropriate control strategy. Chapter 6 focuses on predictive estimation techniques and the justification for selecting an approach capable of addressing system delays and modelling requirements. The implementation and tuning of both the control algorithm and the state estimator, including latency compensation and validation, are presented in Chapter 7. The capabilities of the system are discussed in Chapter 8. Chapter 9 presents the final conclusions. Chapter 10 outlines recommendations for future work.

2 Programme of Requirements

This section defines the requirements for a system designed to track a falling object using a camera mounted on a pan-tilt unit. The goal is to vertically maintain the object within the camera frame throughout the fall by continuously adjusting the orientation of the camera based on the object's position. The system includes a Raspberry Pi 5, a Raspberry Pi HQ camera and a Pimoroni PIM183 pan–tilt unit. The PIM183 pan–tilt actuator receives pulse-width modulated (PWM) commands. Object position is extracted using image processing and actuator commands are computed accordingly. The specifications presented below reflect the known constraints and performance goals.

2.1. Target Environment

The operating environment is a controlled laboratory setup with consistent lighting and a fixed drop zone. The distance of the object from the camera should be known before it falls. The proxy object to be tracked is a coloured ball approximately 5 cm in diameter.

2.2. Functional Requirements

Table 2.1: Mandatory Functional Requirements.

ID	Requirement
MR-01	The system must track a vertically falling object using tilt-only motion, operating at a frequency of 50 FPS.
MR-02	The control loop, consisting of image acquisition, state estimation and command generation, must complete execution within 20 ms.
MR-03	The actuator commands must remain within mechanical tilt constraints of $-90°$ to $+90°$ throughout operation.
MR-04	The system must execute all computations exclusively on the Raspberry Pi 5 without external assistance.

2.3. Trade-Off Requirements

Table 2.2: Trade-Off Requirements.

ID	Requirement
ToR-01	The object's position should remain within $\pm$5% of the image center during the fall.
ToR-02	The control system should minimize overshoot and oscillations in the actuator response.

2.4. Development Constraints

- The system is built on a Raspberry Pi 5 running Raspberry Pi OS.
- The camera is a Raspberry Pi HQ Camera connected via the CSI interface.
- The pan–tilt mechanism is controlled via PWM signals generated by the microcontroller on the PIM183 Pan-Tilt HAT.
- The PWM response delay of the actuator is approximately 20 ms, based on initial hardware documentation.

3 System Characterization

3.1. System Overview

The tracking system was designed to follow and centre a falling object along the vertical axis. For initial system verification, a coloured splash ball was used as a proxy target. The complete control loop consisted of visual detection, which provided the object's centre position along the image's vertical axis; a controller for adjusting the camera's tilt; and state estimation using a filter to feed future predicted positions to the controller, compensating for system delay. The group responsible for object detection [12] only provided y-coordinates. Therefore, the system was limited to vertical tracking. The system overview can be seen in Fig. 3.1.

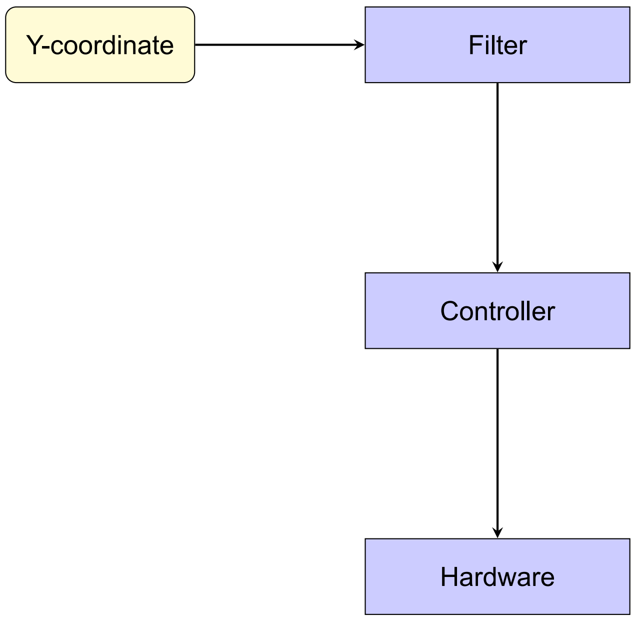

Figure 3.1: Pipeline flowchart of the tracking system.

3.2. Hardware Configuration

To implement a real-time vertical tracking system with predictive control, specific hardware components were selected to meet the processing and actuation requirements under strict timing constraints. Each component played a distinct role within the control loop: acquiring visual information, performing onboard computation and actuating the camera's tilt. The system consisted of the following main components: the Raspberry Pi 5 as the central processing unit, the Raspberry Pi High Quality Camera for visual input and the PIM183 Pan-Tilt HAT for tilt actuation. These components were chosen for their compatibility with the Raspberry Pi platform and low cost. The overall setup is shown in Figs. 3.2 and 3.3.

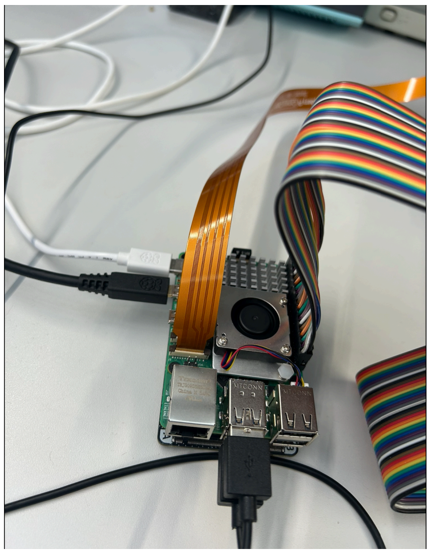

Figure 3.2: Raspberry Pi 5 setup.

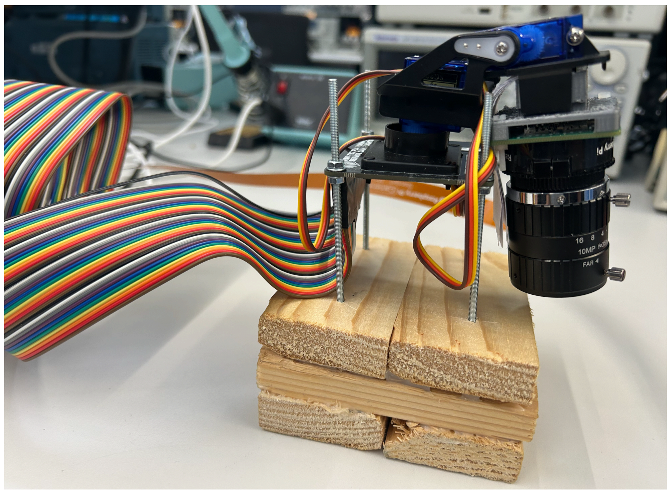

Figure 3.3: PIM183 actuator setup.

3.2.1. Raspberry Pi 5

The Raspberry Pi 5 serves as the central processing unit. It executed the entire tracking pipeline, including image capture, object detection, filter-based prediction and control, at 50 Hz. The board's processing capabilities enable concurrent execution of computationally intensive tasks, while GPIO access allows direct communication with the servo controller. This hardware configuration facilitated a responsive tracking platform, within the physical and computational constraints of embedded systems. The component is illustrated in Fig. 3.4.

Figure 3.4: Raspberry Pi 5.

3.2.2. Raspberry Pi High Quality Camera

The Raspberry PI HQ Camera delivers video frames with a vertical resolution of 640 pixels at a frame rate of 120 FPS. The large sensor enhances image clarity under moderate lighting conditions, while the manual fixed-focus lens offers flexibility to adjust the focal length prior to operation. Fig. 3.5 shows the camera module.

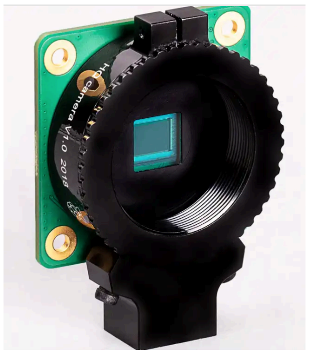

Figure 3.5: Raspberry Pi High Quality Camera.

3.2.3. PIM183 Pan-Tilt HAT

The PIM183 Pan-Tilt HAT was used to actuate the tilt motion of the HQ camera. Only the tilt axis was utilized in this project to follow the vertically falling object. The PIM183 receives control commands directly from the Raspberry Pi GPIO interface. Further details of this module are provided in Chapter 4. The component is shown in Fig. 3.6.

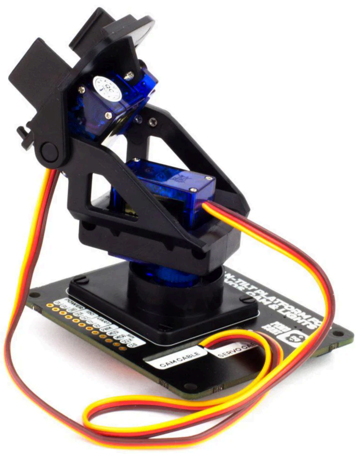

Figure 3.6: PIM183 Module.

4
PIM183 Pimoroni Pan-Tilt HAT

4.1. Actuator Objective

To track a free-falling object in real time, it was necessary to move the camera dynamically. A static camera would lose sight of the target within a fraction of a second, because of its fixed field of view and the high velocity of the object. Since the object's motion is driven by gravity and accelerates rapidly, keeping it in view required the camera to tilt downward in real time to follow the object's fall. This made an actuator essential to adjust the camera angle continuously and keep the object centred in the video frame. Without this capability, the tracking system would fail to maximize visible tracking duration and precision.

The Pimoroni PIM183 Pan-Tilt HAT (hereafter referred to as the PIM183) was selected, because it provided an integrated, compact and Pi-compatible solution for actuating the camera. The HAT features two SG90 micro servos, one for panning and one for tilting. The PIM183 served as the physical actuator in the system's control loop by receiving angular displacement commands to correct the camera's orientation in response to perceived object movement.

4.2. Physical Setup

During the integration phase of the PIM183, several mechanical constraints were encountered. The active cooling system of the Raspberry Pi 5 obstructed direct mounting of the PIM183 on the GPIO pins. This was resolved using a 40-pin male-to-female jumper cable. This jumper cable can be seen in Fig. 4.1.

Another problem was that the actuator experienced a large force when attempting to tilt the camera at full speed. This affected mechanical stability and led to unintended oscillations. To tackle this, long bolts were used to attach the system to a block of wood. This solution significantly increased the stability of the actuator and the camera.

An overview of the complete physical setup is shown in Figs. 3.2 and 3.3, which illustrate the connection between the PIM183 and the Raspberry Pi.

Figure 4.1: 40-pin male-to-female jumper cable used to connect the PIM183 to the Raspberry Pi.

4.3. Characteristics and Actuation Behavior

The tilt axis of the PIM183 is actuated using a standard Pulse Width Modulation (PWM) signal, which is generated by the onboard microcontroller of the PIM183 module. The SG90 micro servo, which can be seen in Fig. 4.2, interprets this PWM signal to determine its target angle within a nominal range of approximately $\pm 90^\circ$.

Figure 4.2: SG90 micro servo.

The control signal operates at a fixed frequency of 50 Hz, which corresponds to one 20 ms cycle per control update. Within each cycle, the pulse width determines the desired angle:

- 1.0 ms pulse $\rightarrow \approx$ 0° (minimum position),
- 1.5 ms pulse $\rightarrow \approx$ 90° (neutral),
- 2.0 ms pulse $\rightarrow \approx$ 180° (maximum position).

This results in a linear mapping between pulse width and angular position, where each microsecond of pulse duration corresponds to approximately 0.09° of rotation. The servo reads this pulse once every 20 ms and begins moving toward the corresponding target angle.

One key limitation in high-speed servoing is the latency introduced by the servo's mechanical response. According to the datasheet, the SG90 operates with a control signal period of 20 ms [13]. This implies that it may take up to one full cycle, 20 ms, for the PIM183 to receive and respond to a new tilt command.

5

Controller Methodology

5.1. Control Objective

The objective of the project was to keep the falling target object centred and within the camera frame throughout its fall. To enable this, a closed-loop control system was required. In the absence of feedback, the system would be incapable of compensating for the fall of the object. Therefore, the platform had to be provided with continuous corrective commands to ensure proper centring during the fall.

5.2. Review of Candidate Control Strategies

Successful real-time tracking requires a control strategy capable of rapid, accurate responses while operating within the computational constraints of the used hardware. To identify the most suitable approach for this project, several prominent control methodologies were evaluated for their applicability to high-speed visual servoing scenarios. These included classic proportional-integral-derivative (PID) control, more advanced model predictive control (MPC) and a brief consideration of other sophisticated techniques.

5.2.1. PID Control

The PID controller is widely used in industrial and embedded applications due to its simple structure and reliable performance across a variety of control problems [14, pp. 1–2]. It combines proportional action for immediate error correction, derivative action to dampen system response and integral action to eliminate steady-state error [14, p. 72]. Although the set-point in image coordinates remains fixed at the centre of the frame, the continuously falling object generates a rapidly changing error signal. As a result, the integral term becomes ineffective and potentially detrimental, since it integrates non-stationary error. This could lead to overshoot and degraded transient response [14, p. 77].

5.2.2. Model Predictive Control

Model Predictive Control is an advanced control approach that optimizes future control actions over a defined time horizon by solving a constrained optimization problem at each time step [15]. Its predictive capability is advantageous in dynamic scenarios, particularly when delays or constraints are present [9]. Nonetheless, MPC typically demands significant computational resources and requires a reliable system model, which may be difficult to construct and maintain on lightweight embedded platforms such as the Raspberry Pi 5. For instance, Forgione et al. [16, p. 5193] reported an average MPC computation time of 22 ms per iteration on a Raspberry Pi 3. This highlights the computational limitations of such devices in real-time settings.

5.2.3. Other Approaches Considered

Additional methods such as Linear Quadratic Regulation [14] and fuzzy control schemes [17] were also acknowledged during the design phase. These methods, while promising in theory, were deemed too complex or model-dependent for the scope and constraints of this application.

5.3. Selection and Rationale

Given the need for responsiveness, computational efficiency and minimal tuning complexity, a Proportional-Derivative (PD) controller was selected. Unlike the full PID approach, the exclusion of the integral term eliminates the risk of integral wind-up caused by the continuously and rapidly changing error signal associated with a falling object. The PD structure provides immediate corrective action through the proportional term while improving system stability and transient response with derivative damping. This configuration was found to be more suitable for high-speed tracking applications, where fast response and stability outweigh steady-state accuracy. Additionally, the PD controller offered a better fit for real-time implementation on embedded hardware with constrained computational resources.

6 Predictive State Estimation

6.1. Estimation Objective

To counteract the delay of up to 20 ms in the PIM183, a prediction-based approach is essential. This delay, introduced by the PWM signal generation, results in control actions based on outdated positional data. Therefore, a real-time state estimator capable of predicting the near-future position of the falling object was required. The estimator must model the trajectory of the falling target, process noisy visual data and be compatible with the computational constraints of the Raspberry Pi 5.

6.2. Review of Candidate Estimation Techniques

Several estimation techniques were reviewed to determine their suitability for real-time object tracking. The methods considered include the classical Kalman Filter (KF), the Extended Kalman Filter (EKF) and the Particle Filter (PF). Each technique is briefly described below.

6.2.1. Kalman Filter

The Kalman Filter is a recursive state estimator that assumes linear system dynamics and additive Gaussian noise in both the process and measurement models [18]. It operates through two main, iterative phases. First, a prediction step uses a known control input and a dynamic model to forecast the system's current state. This is followed by an update step, which then corrects and refines this state estimate based on new measurements. While highly efficient and optimal for linear systems, the KF is limited by its assumption of linearity. In scenarios involving non-linear dynamics, such as gravitational acceleration with air resistance, its accuracy degrades.

6.2.2. Extended Kalman Filter

The Extended Kalman Filter generalizes the standard Kalman Filter to non-linear systems by linearizing the process and observation models around the current state estimate using first-order Taylor expansion [19]. This allows it to estimate states in systems where the motion model includes non-linearities, such as quadratic drag during free fall. The EKF maintains the recursive prediction-update structure of the KF but incorporates Jacobians to approximate the local behaviour of the system. It is particularly suitable for systems with smooth non-linearities and modest computational budgets. EKFs are widely used in object modelling, robot control, target tracking and surveillance due to their efficient handling of mild non-linearities in dynamic systems [20].

6.2.3. Particle Filter

The Particle Filter is a Bayesian estimation method that approximates the posterior distribution of a system's state using a set of weighted particles [21]. Each particle represents a potential state and its weight reflects how well that state explains the observed measurements. Unlike Kalman-based methods, the Particle Filter does not assume linearity or Gaussian noise. This makes it suitable for highly non-linear and multimodal distributions. However, its computational complexity grows with the number of particles required to achieve sufficient estimation accuracy, especially at high update rates such as 50 frames per second. This makes the method difficult to implement in real time on constrained embedded platforms. As noted by Mirzaei et al., while the Particle Filter resolves the limitations of linear and unimodal assumptions, its increased computational cost renders it unsuitable for real-time applications [10, p. 14].

6.3. Selection and Rationale

The Extended Kalman Filter was selected as the optimal state estimation technique due to its superior balance of non-linear predictive performance and computational efficiency for this application. This allows accurate prediction of vertical position and velocity, incorporating both gravity and drag effects. The EKF's recursive nature enables operation at high update rates with minimal latency, making it well-suited for compensating the 20 ms servo delay. Furthermore, its lightweight computational footprint ensures real-time compatibility with the constraints of the Raspberry Pi 5, without compromising estimation fidelity.

7 Prototype Implementation and Validation Results

7.1. PD Controller

A PD controller was implemented to correct the positional offset between the tracked object and the centre of the camera frame. The block diagram in Fig. 7.1 represents the core structure of the PD controller applied to the tracking problem. This configuration is designed to minimize the tracking error by combining a proportional and derivative response before commanding the tilt adjustment. The mathematical formulation of this controller is presented next.

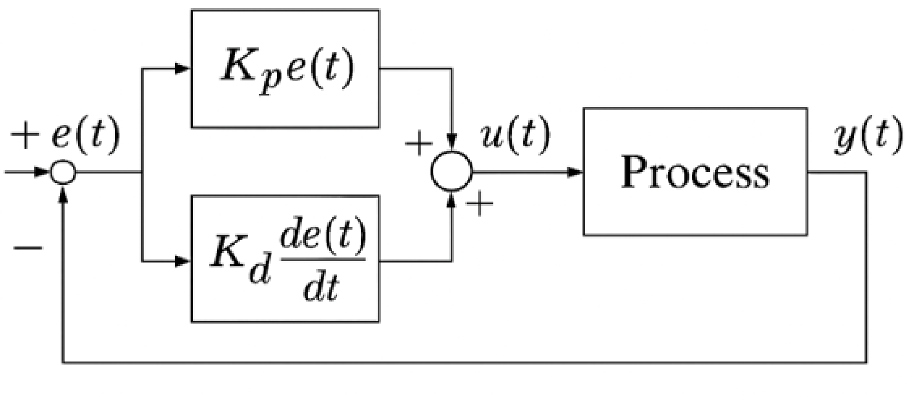

Figure 7.1: Block diagram of the PD controller.

7.1.1. Control Law Formulation

The discrete-time PD control law used in the implementation is expressed as:

$$u(t) = K_p \cdot e(t) + K_d \cdot \frac{de(t)}{dt} \tag{7.1}$$

where $u(t)$ denotes the commanded tilt adjustment in pixels, $e(t)$ is the pixel error between the object's position and the camera's centreline and K_p, K_d are the proportional and derivative gains, respectively [14, p. 64].

7.1.2. Derivative Term with Low-Pass Filtering

To prevent excessive amplification of high-frequency noise, the derivative term in the PD controller was implemented with a first-order low-pass filter. As discussed in [14, p. 73], the ideal derivative $D(s) = K_d s$ has unbounded gain as frequency increases. This is undesirable in practical systems where sensor measurements are noisy. To address this, the derivative term is implemented as:

$$D(s) = \frac{K_d s}{1 + sT_f} \tag{7.2}$$

where $T_f = \frac{T_d}{N}$ is the filter time constant and N is a tuning parameter typically chosen in the range $2 \leq N \leq 20$ to balance noise suppression and responsiveness.

Equation (7.2) represents a standard first-order low-pass filtered derivative, which approximates the ideal derivative at low frequencies while limiting the gain at high frequencies. To realize this in discrete time, the following steps are taken:

- The raw discrete derivative is computed using a backward difference:

$$d_{\text{raw}}(t) = \frac{e(t) - e(t-1)}{dt}$$

- A first-order infinite impulse response (IIR) filter is then applied:

$$d_{\text{filt}}(t) = \alpha \cdot d_{\text{raw}}(t) + (1 - \alpha) \cdot d_{\text{filt}}(t-1)$$

  where the smoothing factor α is defined by:

$$\alpha = \frac{dt}{T_f + dt}$$

This yields the final discrete-time expression:

$$d_{\text{filt}}(t) = \frac{dt}{T_f + dt} \cdot \frac{e(t) - e(t-1)}{dt} + \left(1 - \frac{dt}{T_f + dt}\right) \cdot d_{\text{filt}}(t-1) \tag{7.3}$$

Equation (7.3) matches the implementation used in this work. It ensures that the derivative action remains responsive to meaningful error changes while attenuating high-frequency noise, in accordance with the design recommendations in [14, p. 73].

7.1.3. Actuator Command Generation

Since the object position was expressed in pixel coordinates, the control signal $u(t)$ was initially represented in pixels. To actuate the tilt mechanism, this pixel displacement was converted into an angular command in degrees. The conversion required calculating the camera's vertical angular resolution, defined as the number of degrees per pixel.

The vertical field of view angle (θ_{fov}) was determined using the formula [22, p. 14]:

$$\theta_{\text{fov}} = 2 \cdot \arctan\left(\frac{H_{\text{sensor}}}{2f}\right) \tag{7.4}$$

where H_{sensor} is the sensor height and f the focal length, both in millimetres. The angular resolution was then calculated as:

$$\text{deg/px} = \frac{\theta_{\text{fov}} \cdot 180°/\pi}{H_{\text{px}}} \tag{7.5}$$

where H_{px} is the vertical resolution in pixels. This formulation is based on the pinhole camera model [22, p. 14], a standard in vision-based control systems due to its simplicity and practical accuracy under narrow field-of-view conditions. The camera specifications were based on the Raspberry Pi HQ Camera module, featuring a sensor height of 4.712 mm and a mounted lens with a 35 mm focal length [23]. These parameters result in a vertical field of view (FoV) of approximately 7.7° and an angular resolution of 0.012° per pixel for an image height of 640 pixels.

7.1.4. Gain Tuning Methodology

The proportional and derivative gains (K_p, K_d) of the PD controller were determined through iterative testing using a slow-moving coloured ball. This setup enabled controlled evaluation of the system's response to gradual displacements without rapid dynamics or tracking noise dominating the behavior.

The tuning procedure followed a classical heuristic approach: the proportional gain K_p was gradually increased from zero until the system began to exhibit visible oscillations in response to positional errors. Once continuous oscillation was observed, the value of K_p was halved to achieve a more stable and damped response [24, p. 234].

After determining a suitable K_p, the derivative gain K_d was incrementally adjusted to further reduce overshoot and damp residual oscillations, without significantly slowing the response. The final gain values were selected based on qualitative observation of smooth, stable object tracking under slow motion conditions.

7.2. Extended Kalman Filter

To counteract significant system latency arising from image processing and actuator response, an Extended Kalman Filter (EKF) was implemented. The EKF serves a dual purpose: it filters noise from the visual sensor measurements and provides a predictive estimate of the falling object's state. This enables a feed-forward control strategy that targets the object's future position, rather than reacting to delayed and noisy observations.

7.2.1. System Modeling

The state of the system at time step k is defined by the vector:

$$\mathbf{x}_k = \begin{bmatrix} y_k \\ \dot{y}_k \end{bmatrix} \tag{7.6}$$

where y_k is the vertical position and $\dot{y}_k$ is the vertical velocity in the image frame.

The system dynamics are described by a non-linear process model that accounts for gravity and quadratic air drag:

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}) + w_{k-1} \tag{7.7}$$

where f is the non-linear state transition function and w_{k-1} is a zero-mean Gaussian process noise with covariance $\mathbf{Q}$.

To apply a physics-based model, physical parameters were scaled into the pixel domain. A scale factor $s_{\text{p/m}}$ (in px/m) was determined from the camera's geometry using the pinhole model [22]:

$$s_{\text{p/m}} = \frac{H_{\text{px}}}{2D \cdot \tan\left(\frac{\theta_{\text{fov}}}{2}\right)} \tag{7.8}$$

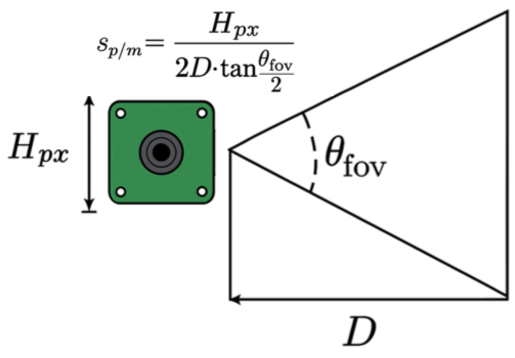
$s_{p/m} = \frac{H_{px}}{2D \cdot \tan\frac{\theta_{\text{fov}}}{2}}$
H_{px}
θ_{fov}
D

Figure 7.2: Camera geometry for determining pixel-to-meter scale factor using the pinhole model.

where H_{px} is the sensor's vertical resolution in pixels, D is the known object distance in meters and θ_{fov} is the vertical field of view.

Consequently, gravitational acceleration $g = 9.81\,\text{m/s}^2$ was converted to pixel units as $g_{\text{pix}} = g \cdot s_{\text{p/m}}$. The drag term, based on the standard drag equation, was also converted to a pixel-based coefficient to enable its use in image coordinate space. The resulting proportional gain K_p retains the dimension of inverse meters:

$$[K_p] = \mathrm{m}^{-1}$$

This was subsequently scaled by a pixel-to-meter conversion factor to match the coordinate system used by the vision-based tracking system,

$$k_p = \frac{0.5 \cdot \rho \cdot C_d \cdot A}{m \cdot s_{\text{p/m}}} \tag{7.9}$$

where ρ is air density, C_d is the drag coefficient, A is the cross-sectional area and m is the object mass.

7.2.2. EKF Prediction and Update Cycle

The EKF operates in a two-step cycle:

1. Prediction: The state and covariance are projected forward in time. The **predicted** state estimate $\mathbf{x}_k^-$ is computed using the non-linear model:

$$a_{\text{net}} = g_{\text{pix}} - k_p \cdot (\dot{y}_{k-1})^2 \tag{7.10}$$

$$y_k^- = y_{k-1} + \dot{y}_{k-1}\Delta t + \frac{1}{2}a_{\text{net}}\Delta t^2 \tag{7.11}$$

$$\dot{y}_k^- = \dot{y}_{k-1} + a_{\text{net}}\Delta t \tag{7.12}$$

The system dynamics are linearized around the previous state estimate $\mathbf{x}_{k-1}$ to compute the Jacobian matrix $\mathbf{F}_k$:

$$\mathbf{F}_k = \begin{bmatrix} 1 & \Delta t - k_p \cdot \dot{y}_{k-1} \cdot \Delta t^2 \\ 0 & 1 - 2 \cdot k_p \cdot \dot{y}_{k-1} \cdot \Delta t \end{bmatrix} \tag{7.13}$$

The **predicted** covariance estimate is then found by:

$$\mathbf{P}_k^- = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^\top + \mathbf{Q}_{k-1} \tag{7.14}$$

2. Update: The **predicted** estimate is corrected using the incoming measurement z_k. The measurement model is linear, observing only the position:

$$z_k = \mathbf{H}\mathbf{x}_k + v_k, \quad \text{with } \mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{7.15}$$

where v_k is zero-mean Gaussian measurement noise with covariance R. The standard EKF update equations are then applied [19]:

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}^\top (\mathbf{H}\mathbf{P}_k^- \mathbf{H}^\top + R)^{-1} \tag{7.16}$$

$$\mathbf{x}_k = \mathbf{x}_k^- + \mathbf{K}_k (z_k - \mathbf{H}\mathbf{x}_k^-) \tag{7.17}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H})\mathbf{P}_k^- \tag{7.18}$$

where $\mathbf{K}_k$ is the optimal Kalman gain and $\mathbf{x}_k$ and $\mathbf{P}_k$ are the final **updated** state and covariance estimates for time step k.

7.2.3. Latency Compensation

A key function of the filter is to provide a future state prediction, $\hat{\mathbf{x}}_{k+A|k}$, to compensate for a total system latency of approximately 20 ms. This prediction was generated by extrapolating the non-linear dynamics model over a fixed time horizon as shown in Equations (7.10)–(7.12), for A future time steps, starting from the latest updated state $\mathbf{x}_k$. As the EKF ran at 50 Hz, A was set to 1 to predict exactly 20 ms ahead. This physics-informed projection yields an accurate target for the controller, effectively mitigating the delay.

This EKF framework fuses noisy sensor data with a physical process model to produce robust state estimates. Its predictive capability is critical for achieving real-time control in the presence of systemic latencies, while remaining computationally efficient for on-board processing [19].

7.3. Experimental Validation and System Tuning

7.3.1. Initial Trials and Problem Identification

Initial validation trials revealed that the system was unable to reliably track the falling object. In early experiments, the camera would not follow the ball during free fall and instead exhibited a single, small corrective movement only after the ball had already impacted the ground. This indicated a substantial latency between the visual detection, state estimation and actuator response.

7.3.2. Debugging and Latency Analysis

To methodically isolate the source of the delay, the investigation began with the actuator itself. A minimal test script was developed to command the PIM183 tilt actuator directly, bypassing the entire object detection, EKF and PD control pipeline. This allowed for the characterization of the actuator's baseline performance in isolation. To capture the results, a separate Raspberry Pi AI camera was set up as an independent observer, recording the actuator's response to step commands under controlled conditions. The actuator was given a step command to change tilt angle and a red visual marker was displayed immediately upon issuing the command to mark the frame of initiation. The scene was recorded at 30 frames per second under two conditions: (1) with the HQ camera mounted on the actuator and (2) without it.

In each condition, multiple trials were recorded and the resulting frame sequences were analysed. The delay between the command signal and the first visible actuator movement was measured in frames. It was observed that without the camera mounted, the actuator responded after approximately two frames (≈66 ms), whereas with the camera mounted, the delay increased to approximately three frames (≈100 ms). This confirmed that the weight introduced by the HQ camera assembly significantly increased the actuator's response latency, but also that the PIM183 itself had a latency which was higher than the expected 20 ms.

Including software processing delays measured during pipeline profiling, the total effective delay of the system was estimated to be approximately 160 ms. The original EKF configuration had assumed a single-frame prediction horizon. To account for the newly measured delay, the EKF was reconfigured to predict eight frames ahead. This modification allowed the controller to target a future position of the object, partially mitigating the latency.

7.3.3. Results

Subsequent tests with the revised EKF showed improved tracking performance. Frame-by-frame video analysis revealed that the ball remained within the camera's field of view for approximately 40–50% of its fall duration. However, the object often appeared off-centre and the tracking still lagged during fast downward motion. This behaviour was attributed to the combination of physical limitations of the actuator, residual latency and the camera's mounting instability, which caused vibrations during tilt corrections.

8 Discussion

The primary objective of this project was to achieve real-time centring and tracking of a vertically falling water droplet using embedded, low-cost hardware. While the final system demonstrated partial success in tracking a coloured splash ball within the camera frame, the complete centring objective and the ultimate goal of tracking a water droplet proved unattainable.

The integration of a Proportional–Derivative (PD) controller with an Extended Kalman Filter (EKF) enabled predictive motion control to compensate for significant system latency. The EKF was configured to predict the object's vertical position eight frames ahead, corresponding to a 160 ms delay at 50 frames per second. This predictive control strategy improved the tracking response relative to initial implementations that relied solely on current position data.

However, the mechanical response characteristics of the PIM183 actuator constituted a critical performance bottleneck. Despite the controller's ability to issue commands in real time, the actuator introduced delays of up to 100 ms due to the camera's weight and motor limitations of the PIM183. Analysis confirmed that the system was limited to issuing no more than two effective corrective inputs during the entire fall duration of the object.

The system was not tested on water droplets, as it could not be detected by the object detection algorithm. Since detecting a splash ball was already challenging, it can be assumed that detecting a water droplet, would be even more difficult.

The experiments validated that predictive control with an EKF–PD configuration can meaningfully mitigate latency in systems with modest dynamic demands. However, this approach becomes inadequate when applied to high-speed, low-mass objects with rapid positional changes, such as falling water droplets. The limitations of low-torque actuators and delayed image processing outweigh the benefits of model-based prediction. These hardware constraints prevent the system from reacting at the required rate to track the object accurately. Although the control strategy is theoretically sound and well-designed, its practical effectiveness is restricted. The overall system performance is ultimately bounded by the responsiveness of the hardware and the quality of the visual measurements.

9 Conclusion

This thesis investigated the design and implementation of a real-time motion control system for vertically tracking a free-falling object using embedded hardware. The primary aim was to maintain a water droplet centred in the camera frame throughout its descent. Due to its limited visibility and rapid motion, a coloured splash ball was initially used as a proxy to validate the concept. Despite this simplification, the complete centring objective remained out of reach.

The final system employed a Proportional–Derivative (PD) controller together with an Extended Kalman Filter (EKF). This configuration aimed to counteract substantial actuation and processing latency, which was experimentally determined to be approximately 160 ms. The EKF predicted the target's future position eight frames ahead, providing the PD controller with a forward-looking reference. This approach improved system responsiveness compared to non-predictive implementations, allowing the ball to remain within the camera frame for nearly half of its fall duration.

Nevertheless, several key limitations emerged. The PIM183 actuator exhibited a mechanical delay of up to 100 ms, compounded by the inertia of the mounted camera. This restricted the control loop to no more than two effective adjustments during the fall, making precise tracking unfeasible. Water droplet tracking was not attempted, as it could not be tracked correctly by the object detection.

In summary, the system fell short of the real-time accuracy and agility required for high-speed droplet tracking. The results highlight the need for faster actuators and improved visual detection to enable precise tracking of small objects in real time. Despite these limitations, the system forms a useful foundation for future research into embedded vision-based tracking systems with predictive capabilities.

10 Recommendation and Future Work

The results of this project highlight several promising directions for future research and development in the domain of embedded visual tracking systems.

Firstly, one of the most critical hardware limitations identified was the mechanical delay introduced by the PIM183 actuator. To improve system responsiveness, future designs should adopt higher-torque, low-latency actuators. Alternatives such as direct-drive motors or faster servo mechanisms could significantly reduce the effective system delay and increase the number of corrective actions possible during a fall.

Secondly, the mass and inertia introduced by the Raspberry Pi HQ camera noticeably impacted the tilt system's performance. Employing a lighter camera module or optimising the mechanical mounting could improve system stability and reduce vibrations, leading to more consistent tracking performance.

Furthermore, while the Extended Kalman Filter improved predictive control under latency constraints, the detection system itself struggled with reliability under fast-motion conditions. Future work should explore integrating more advanced object detection techniques, which may offer better performance with small, fast-moving targets such as water droplets.

Finally, although tracking water droplets was beyond the capabilities of the current system, the underlying setup showed promising results when applied to slower-moving objects. This suggests that the overall approach has potential, especially if key limitations are addressed. Future research could focus on improving detection for small and fast targets, for instance by applying adaptive filtering techniques or integrating additional sensors to enhance reliability in visually challenging conditions.

By addressing these mechanical, computational and algorithmic constraints, the system could be developed into a more versatile and responsive tracking solution that is capable of operating in real time and handling high-speed motion with greater precision and stability.

A

Python codes

A.1. PD controller

```python
import time

class PD:
    def __init__(self, kp, kd, setpoint, deg_per_px,
                 tau=0.02, max_dt=0.1):
        # PID gains
        self.kp = kp
        self.kd = kd

        # Desired setpoint in pixels (centre of image height in camera frame)
        self.setpoint = setpoint

        # Conversion factor (amount of degrees a pixel represents)
        self.deg_per_px = deg_per_px

        # Derivative smoothing and dt cap
        self.tau = tau
        self.max_dt = max_dt


        # Internal state
        self.prev_time = time.monotonic()
        self.deriv = 0.0
        self.prev_error = None
        self.last_error = 0.0
        self.last_dt = 1e-6

    def update(self, measurement):
        now = time.monotonic()
        dt = max(1e-6, min(now - self.prev_time, self.max_dt))

        # Compute error in pixels
        error = self.setpoint - measurement

        # Derivative on error, low-pass filtered
        if self.prev_error is None:
            self.deriv = 0.0
        else:
            raw_deriv = (error - self.prev_error) / dt
            alpha = dt / (self.tau + dt)
            self.deriv = alpha * raw_deriv + (1 - alpha) * self.deriv

        # PID terms
        P = self.kp * error
        D = self.kd * self.deriv
        output_px = P + D

        # Save state
        self.prev_time = now
        self.prev_error = error
        self.last_error = error
        self.last_dt = dt

        # Convert pixel-output to degrees
        return -output_px * self.deg_per_px

    def reset(self):
        # Clear integral and derivative history
        self.integral = 0.0
        self.deriv = 0.0
        self.prev_error = None
        self.prev_time = time.monotonic()
        self.last_error = 0.0
        self.last_dt = 1e-6
```

A.2. Extended Kalman Filter

```python
import numpy as np
import math

class EKF_BallTracker:
    def __init__(self,
                 initial_y, initial_vy,
                 initial_P_yy, initial_P_vv,
                 process_noise_y, process_noise_vy,
                 measurement_noise_y,
                 dt, g_pix, k_drag_pix, px_per_m):
        self.px_per_m = px_per_m

        # State vector [y, vy]^T
        self.x_hat = np.array([initial_y, initial_vy], dtype=float)

        # State covariance matrix P_k|k
        self.P = np.diag([initial_P_yy, initial_P_vv]).astype(float)

        # Process noise covariance Q
        self.Q = np.diag([process_noise_y, process_noise_vy]).astype(float)

        # Measurement noise covariance R
        self.R = float(measurement_noise_y)

        # Time step
        self.dt = float(dt)

        # Physical parameters (in pixel units)
        self.g_pix = float(g_pix)
        self.k_drag_pix = float(k_drag_pix)  # Units: 1/pixel

        # Measurement matrix H (constant)
        self.H = np.array([[1, 0]], dtype=float)
        self.H_T = self.H.T

        # Identity matrix
        self._I = np.eye(2)

    def predict(self):
        y_prev, vy_prev = self.x_hat
        a_net = self.g_pix - self.k_drag_pix * vy_prev**2 # Simple quadratic drag model

        # State prediction
        y_pred = y_prev + vy_prev * self.dt + 0.5 * a_net * self.dt**2
        vy_pred = vy_prev + a_net * self.dt
        self.x_hat = np.array([y_pred, vy_pred])

        F = np.zeros((2, 2), dtype=float)
        F[0, 0] = 1.0
        F[0, 1] = self.dt - self.k_drag_pix * vy_prev * self.dt**2 # Use vy_prev from start of
     interval
        F[1, 0] = 0.0
        F[1, 1] = 1.0 - 2.0 * self.k_drag_pix * vy_prev * self.dt # Use vy_prev from start of
    interval

        # Covariance prediction
        self.P = F @ self.P @ F.T + self.Q
        return self.x_hat

    def update(self, measurement_y):
        # Innovation (measurement residual)
        y_tilde = measurement_y - self.x_hat[0] # x_hat[0] is y_pred_k|k-1

        # Innovation covariance
        # S = H P_k|k-1 H^T + R
        # Since H = [1, 0], H P H^T = P[0,0]
        S = self.P[0, 0] + self.R
        if S == 0: # Avoid division by zero
            S = 1e-9 # Add a tiny epsilon

        # Kalman gain K
        # K = P_k|k-1 H^T S^-1
        # P_k|k-1 H^T = [P[0,0], P[1,0]]^T
        K_column_vector = self.P @ self.H_T
        K = K_column_vector / S

        # State update
        self.x_hat = self.x_hat + (K * y_tilde).flatten()

        # Covariance update
        self.P = (self._I - K @ self.H) @ self.P
        # Ensure P remains symmetric
        self.P = 0.5 * (self.P + self.P.T)
        return self.x_hat

    def get_current_position(self):
        return self.x_hat[0]

    def get_current_velocity(self):
        return self.x_hat[1] / self.px_per_m

    def get_predicted(self):
        """
        Predicts next-step position based on the current updated state x_hat (k|k).
        This is y_hat (k+1|k).
        """
        A = 8
        y_curr_updated, vy_curr_updated = self.x_hat # These are x_k|k
        a_net_curr_updated = self.g_pix - self.k_drag_pix * vy_curr_updated**2
        return y_curr_updated + vy_curr_updated * A * self.dt + 0.5 * a_net_curr_updated * (A
    * self.dt)**2
```

A.3. System Integration

```python
import os
import sys
import time
import math

import cv2 as cv
import numpy as np

import processor.algorithms.colored_frame_difference as proc_color
import processor.algorithms.frame_difference as proc_naive
import processor.algorithms.dummy_algorithm as dummy

import pantilthat as pth
import controller.pd as pd_mod
import controller.ekf as ekf_mod

from processor.camera import CameraStream, SharedObject
from threading import Thread


def clamp(value, vmin, vmax):
    return max(vmin, min(vmax, value))


def tilt(shared_obj):
    while True:
        if shared_obj.is_exit:
            sys.exit(0)
        loop_start = time.monotonic()
        pth.pan(0)
        pth.tilt(shared_obj.current_tilt)

        elapsed = time.monotonic() - loop_start
        sleep_time = shared_obj.LOOP_DT_TARGET - elapsed
        if sleep_time > 0:
            time.sleep(sleep_time)



if __name__ == '__main__':
    # Create shared-memory for capturing, processing and tilting
    shared_obj = SharedObject()

    # Initialize camera
    camera = CameraStream(shared_obj)
    camera.start()

    # Frame dimensions and timing
    FRAME_HEIGHT = 640
    FRAME_RATE = 50
    shared_obj.LOOP_DT_TARGET = 1.0 / FRAME_RATE

    # Compute vertical FoV
    SENSOR_HEIGHT_MM = 4.712
    FOCAL_LENGTH_MM = 35
    vfov_rad = 2 * math.atan(SENSOR_HEIGHT_MM / (2 * FOCAL_LENGTH_MM))
    vfov_deg = math.degrees(vfov_rad)
    deg_per_px = vfov_deg / FRAME_HEIGHT
    setpoint = FRAME_HEIGHT / 2

    # PID and servo settings
    SERVO_MIN, SERVO_MAX = -90, 90

    # Initialize PanTilt HAT
    current_tilt = -23
    try:
        pth.servo_enable(2, True)
        pth.tilt(int(current_tilt))
        print("[INFO] Tilt servo initialized.")

    except Exception as e:
        print(f"[ERROR] Could not initialize PanTilt HAT: {e}")
        camera.stop()
        exit()

    # Camera variables
    # Measurement
    camera_preview_output = None
    camera_prev_gray = None

    # FPS Overlay
    camera_prev_time = time.time_ns()
    camera_diff_time = 0
    camera_frame_per_sec = 0
    camera_frame_cnt_in_sec = 0
    camera_is_one_sec_passed = False
    recording_id = time.strftime('%y%m%d%H%M%S', time.gmtime())

    # Colors
    color_hues = {
        "Red": 0,
        "Green": 60,
        "Blue": 120,
        "Cyan": 90,
        "Magenta": 150,
        "Yellow": 30,
        "Amber": 15,
        "Chartreuse": 45,
        "Spring Green": 75,
        "Azure": 105,
        "Violet": 135,
        "Rose": 165
    }

    pd = pd_mod.PD(kp=0.08, kd=0.01, tau=0.02, setpoint=setpoint, deg_per_px=deg_per_px)

    DROP_DIST_M = 4.5
    world_h_m = 2 * DROP_DIST_M * math.tan(vfov_rad / 2)
    px_per_m = FRAME_HEIGHT / world_h_m
    g_pix = 9.81 * px_per_m

    # Ball properties (example values - YOU MUST MEASURE/DETERMINE YOURS)
    mass = 0.005     # kg (example mass of a dry ping pong ball)
    rho_air = 1.2         # kg/m^3 (air density at sea level, 15°C)
    Cd = 0.47               # Drag coefficient for a sphere (typical value)
    radius_m = 0.025        # m (for a 4cm diameter ball, e.g., ping pong ball)
    A_m2 = np.pi * radius_m**2 # Cross-sectional area m^2
    # k_drag_world = 0.5 * rho_air * Cd * A_m2 (units: kg/m)
    # k_drag_pix = k_drag_world / (px_per_m * mass) (units: 1/pixel)
    k_drag_pix = (0.5 * rho_air * Cd * A_m2 / px_per_m) / mass

    # initialize with the first measurement ('well overwrite vy once we get the 2nd sample)
    initial_y   = 0.0
    initial_vy  = 0.0
    P_y0        = 2500.0         # e.g. measurement variance
    P_vy0       = (2*P_y0)/(shared_obj.LOOP_DT_TARGET**2)
    Qy, Qvy     = 900.0, 925.0
    Ry          = 2500.0

    ekf = ekf_mod.EKF_BallTracker(
        initial_y, initial_vy,
        P_y0, P_vy0,
        Qy, Qvy,
        Ry,
        shared_obj.LOOP_DT_TARGET, g_pix, k_drag_pix, px_per_m
    )

    ekf_initialized = False
    prev_meas = None

    print("[INFO] Starting tracking loop. Press Ctrl+C to exit.")
    pth.idle_timeout(FRAME_RATE)
    # iter_machine = dummy.DummyMeasurements()
    try:
        while True:
            loop_start = time.monotonic()

            # 1) Read frame
            current_frame = shared_obj.frame
            current_gray_frame = cv.cvtColor(current_frame, cv.COLOR_RGB2HSV) if current_frame is not None else None

            # 2) Detect object
            if current_frame is None or camera_prev_gray is None:
                measurement_y = None
            else:
                # camera_preview_output, measurement_y = proc_naive.process_frames(camera_prev_gray, current_gray_frame, current_frame)
                measurement_y, camera_preview_output, _ = proc_color.process_frames(camera_prev_gray, current_gray_frame, current_frame, color_hues["Rose"], hue_tolerance=10)
                # measurement_y, camera_preview_output, _ = iter_machine.next(), current_frame, None

            print(f"info: y: {measurement_y}")
            camera_prev_gray = current_gray_frame

            # 2.1) FPS overlay
            camera_frame_cnt_in_sec += 1
            camera_curr_time = time.time_ns()
            camera_diff_time += (camera_curr_time - camera_prev_time) / 1e6

```

```python
        if int(camera_diff_time) >= 1000:
            camera_frame_per_sec = camera_frame_cnt_in_sec
            camera_frame_cnt_in_sec = 0
            camera_diff_time = 0
            camera_is_one_sec_passed = True

        if camera_is_one_sec_passed:
            cv.putText(current_frame, f"FPS: {camera_frame_per_sec}", (10, 25), cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
        else:
            cv.putText(current_frame, f"FPS: (WAITING...)", (10, 25), cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

        camera_prev_time = camera_curr_time

        # 5) PID update and servo write when active
        if measurement_y is not None:
            if not ekf_initialized:
                # bootstrap velocity from first two samples
                if prev_meas is None:
                    prev_meas = measurement_y
                else:
                    initial_vy = (measurement_y - prev_meas) / shared_obj.LOOP_DT_TARGET
                    ekf.x_hat = np.array([measurement_y, initial_vy])
                    ekf_initialized = True
            else:
                # 1) EKF prediction
                ekf.predict()

                # 2) EKF update with the new pixel measurement
                ekf.update(measurement_y)

                # 3) Get the filtered position & velocity
                y_filt  = ekf.get_current_position()
                vy_filt = ekf.get_current_velocity()

                # 4) Predict next-step y for your PID set-point
                y_next_pred = ekf.get_predicted()

                if abs(setpoint - y_next_pred) > 0:
                    cv.line(current_frame, (0, int(setpoint)), (current_frame.shape[1], int(setpoint)), (0, 0, 255), 2)
                    delta_deg = pd.update(y_next_pred)
                else:
                    delta_deg = 0.0

                desired = current_tilt + delta_deg
                current_tilt = clamp(desired, SERVO_MIN, SERVO_MAX)
                print(f"info: tilt: {current_tilt} deg")

        if current_frame is not None:
            cv.imshow(f'[{recording_id}] [Live] Processed Frame', current_frame)

        pth.tilt(current_tilt)

        # 6) Exit & Store frames
        if cv.waitKey(1) & 0xFF == ord('q'):
            shared_obj.is_exit = True

            output_dir = os.path.join("output_frames", recording_id)
            os.makedirs(output_dir, exist_ok=True)

            for i, frame in enumerate(shared_obj.frame_buffer):
                filename = os.path.join(output_dir, f"frame_{i:06d}.png")
                print(f'info: storing frames [{i:06d}/{len(shared_obj.frame_buffer)}]')
                cv.imwrite(filename, frame)
            sys.exit(0)

        # Fix FPS
        elapsed = time.monotonic() - loop_start
        sleep_time = shared_obj.LOOP_DT_TARGET - elapsed
        if sleep_time > 0:
            time.sleep(sleep_time)
```

```python

    except KeyboardInterrupt:
        print("\n[INFO] Exiting, disabling tilt servo.")

    finally:
        pth.servo_enable(2, False)
        camera.stop()
        cv.destroyAllWindows()
```

Bibliography

[1] J. T. Zhou, J. Du, H. Zhu, X. Peng, Y. Liu, and R. S. M. Goh, "AnomalyNet: An Anomaly Detection Network for Video Surveillance," *IEEE Transactions on Information Forensics and Security*, vol. 14, pp. 2537–2550, 2019. DOI: 10.1109/TIFS.2019.2900907.

[2] H. Zhou, L. Wei, C. P. Lim, D. Creighton, and S. Nahavandi, "Robust Vehicle Detection in Aerial Images Using Bag-of-Words and Orientation Aware Scanning," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 12, pp. 7074–7085, 2018. DOI: 10.1109/TGRS.2018.2848243.

[3] F. Chaumette and S. Hutchinson, "Visual servo control. i. basic approaches," *Robotics Automation Magazine, IEEE*, vol. 13, pp. 82–90, Jan. 2007. DOI: 10.1109/MRA.2006.250573.

[4] B. J. Nelson and P. K. Khosla, "Visually servoed manipulation using an active camera," in *Proceedings of the 33rd Annual Allerton Conference on Communication, Control, and Computing*, [Online]. Available: https://www.ri.cmu.edu/pub_files/pub3/nelson_bradley_1995_2/nelson_bradley_1995_2.pdf, Urbana-Champaign, IL, Oct. 1995.

[5] Q.-Y. Gu and I. Ishii, "Review of some advances and applications in real-time high-speed vision: Our views and experiences," *International Journal of Automation and Computing*, vol. 13, no. 4, pp. 305–318, 2016, [Online]. Available: https://doi.org/10.1007/s11633-016-1024-0. DOI: 10.1007/s11633-016-1024-0.

[6] K. Okumura, K. Yokoyama, H. Oku, and M. Ishikawa, "1ms auto pan–tilt– video shooting technology for objects in motion based on Saccade Mirror with background subtraction," *Advanced Robotics*, vol. 29, no. 7, pp. 457–468, 2015. DOI: 10.1080/01691864.2015.1011299.

[7] B. Torkaman and M. Farrokhi, "Real-time visual tracking of a moving object using pan and tilt platform: A kalman filter approach," in *Proc. 20th Iranian Conference on Electrical Engineering (ICEE)*, 2012, pp. 928–933. DOI: 10.1109/IranianCEE.2012.6292486.

[8] E. Yılmazlar and H. Kuşçu, "Object tracking by pid control and image processing on embedded system," *Research Inventy: International Journal of Engineering And Science*, vol. 6, no. 9, pp. 33–37, 2017, ISSN (e): 2278-4721, ISSN (p): 2319-6483. [Online]. Available: https://www.researchgate.net/publication/321996553_Object_Tracking_by_PID_Control_and_Image_Processing_On_Embedded_System.

[9] R. Nebeluk, K. Zarzycki, D. Seredyński, *et al.*, "Predictive tracking of an object by a pan–tilt camera of a robot," *Nonlinear Dynamics*, vol. 111, pp. 8383–8395, 2023. DOI: 10.1007/s11071-023-08295-z.

[10] B. Mirzaei, H. Nezamabadi-pour, A. Raoof, and R. Derakhshani, "Small object detection and tracking: A comprehensive review," *Sensors*, vol. 23, no. 15, p. 6887, 2023. DOI: 10.3390/s23156887. [Online]. Available: https://www.mdpi.com/1424-8220/23/15/6887.

[11] K. Schulte, C. Tropea, and B. Weigand, "Droplet dynamics under extreme ambient conditions," in *Fluid Mechanics and Its Applications*, vol. 124, Springer, 2022, pp. 1–27.

[12] M. E. Kaya, T. A. Haas, and N. E. Khatibi, "Falling droplet localization: Visual recording object oriented mapping," B.Sc. thesis, Delft University of Technology, Delft, Netherlands, Jun. 2025.

[13] Tower Pro, *Micro servo 9g SG90 - Datasheet*, https://www.kjell.com/globalassets/mediaassets/701916_87897_datasheet_en.pdf, 2024.

[14] K. J. Åström and T. Hägglund, *Advanced PID Control*. Research Triangle Park, NC: ISA - The Instrumentation, Systems, and Automation Society, 2006, ISBN: 1-55617-942-1.

[15] P. Chaber, W. Liu, J. Wu, and D. Li, "Model predictive tracking control for a 2-dof gimbal system based on object motion prediction," *Nonlinear Dynamics*, vol. 113, no. 20, pp. 21 327–21 344, 2023. DOI: 10.1007/s11071-023-08295-z. [Online]. Available: https://doi.org/10.1007/s11071-023-08295-z.

[16] M. Forgione, D. Piga, and A. Bemporad, “Efficient calibration of embedded mpc,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 5189–5194, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: `https://doi.org/10.1016/j.ifacol.2020.12.1188`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2405896320315822`.

[17] Y. L. Melese, G. K. Alitasb, and M. D. Belete, “Optimal fuzzy-pid controller design for object tracking,” *Scientific Reports*, vol. 15, p. 12 064, 2025. DOI: `10.1038/s41598-025-92309-w`. [Online]. Available: `https://www.nature.com/articles/s41598-025-92309-w`.

[18] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[19] A. Becker, *Kalman Filter from the Ground Up*. Alex Becker, 2023, Available: `https://ebooknice.com/product/kalman-filter-from-the-ground-up-50262730`, ISBN: 978-9655984392.

[20] S. Y. Chen, “Kalman filter for robot vision: A survey,” *IEEE Transactions on Industrial Electronics*, vol. 59, no. 11, pp. 4409–4420, 2012. DOI: `10.1109/TIE.2011.2162714`.

[21] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, “A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking,” *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002. DOI: `10.1109/78.978374`.

[22] E. Gamy, *Camera sensor size fov (3)*, [Online], Appendix: “How to calculate angle of view”, 2015. [Online]. Available: `https://www.academia.edu/34473790/Camera_Sensor_Size_FOV_3_`.

[23] Raspberry Pi Foundation, *Raspberry pi hq camera module*, [Online], Accessed: Jun. 15, 2025. Available: `https://www.raspberrypi.com/products/raspberry-pi-high-quality-camera/`, 2020.

[24] K. J. Åström, *Control system design: Chapter 6 – pid control*, `https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/astrom-ch6.pdf`, Accessed June 2025, 2002.