Department of Mechatronics Engineering
Faculty of Engineering and Materials Science
German Universitay in Cairo

Design and Control of a Ball Balancing Plate

MCTR601 Mechatronics Engineering
Supervised By:
Prof. Ayman A. El-Badawy

Mostafa Mohamed Abdelaziz 58-22509
Marwan ElSayed 58-20308
Ali Emad 58-13891

Group Number: T34

Contents

Abstract

This project presents the design, implementation, and analysis of a real-time dynamic ball balancing system with two degrees of freedom. The system employs a camera-based vision sensor and multiple control strategies—PID, a custom-designed PD controller (PV Controller), and Linear Quadratic Regulator (LQR)—to stabilize a ball on a flat platform by continuously adjusting its tilt along the x and y axes. The platform is actuated by two orthogonally placed servo motors, with control executed by an STM32F103C8T6 Bluepill microcontroller. A key feature of the system is trajectory tracking, allowing the ball to follow predefined paths with precision.

The hardware includes a tiltable platform, servo motors, and a wireless camera module that streams real-time images to a processing unit. Image processing algorithms extract the ball's position, calculate its deviation from the target, and generate appropriate control signals. These signals are transmitted to the microcontroller via Bluetooth, enabling online tuning of control parameters such as Kp, Ki, Kd, and dynamic selection of control modes.

The software pipeline integrates image acquisition, preprocessing, ball detection, coordinate estimation, control signal generation, and actuation. Electrical wireless connections ensure seamless communication between components, enabling synchronized operation across hardware and software layers.

Experimental results validate the system's ability to maintain stability and accurately follow trajectories under different control algorithms. The project highlights the effective integration of embedded systems, computer vision, and control theory.

Introduction

In the ever-evolving fields of robotics and automation, achieving precise control and dynamic stability remains a core challenge. Among the systems designed to tackle this challenge, ball balancing platforms stand out as an elegant demonstration of real-time control, sensor integration, and system responsiveness. This project explores the development of a two-degree-of-freedom ball balancing system that combines computer vision and advanced control strategies to maintain equilibrium on a flat, tiltable surface.

The primary goal is to design a system that can dynamically stabilize a rolling ball by adjusting the tilt of a platform in both the x and y directions. Accomplishing this requires seamless integration between mechanical actuation and sensory feedback. The system employs servo motors to manipulate platform orientation, while a camera sensor provides continuous visual feedback of the ball's position. Image processing algorithms analyze the visual data to determine positional errors, which are then corrected using control algorithms such as PID, PD, and LQR.

By uniting hardware components with real-time software processing, the project not only demonstrates the principles of feedback control but also serves as a practical platform for testing and comparing different control approaches under real-world conditions.

System Overview

The ball balancing system integrates a suite of hardware and software components working cohesively to achieve real-time stabilization of a ball on a two-dimensional tilting platform. The project encapsulates four key elements: a user interface, mechanical setup, image processing pipeline, and embedded control logic.

1. Java-Based Graphical User Interface (GUI):

• A custom-designed, JavaFX-powered GUI provides users with full control over the system. This user-friendly interface enables:

• Mode Selection: Choose between multiple operating modes such as idle, automatic, manual, or calibration.

• Control Algorithm Switching: Dynamically select between PID, PD (PV), LQR, or custom algorithms.

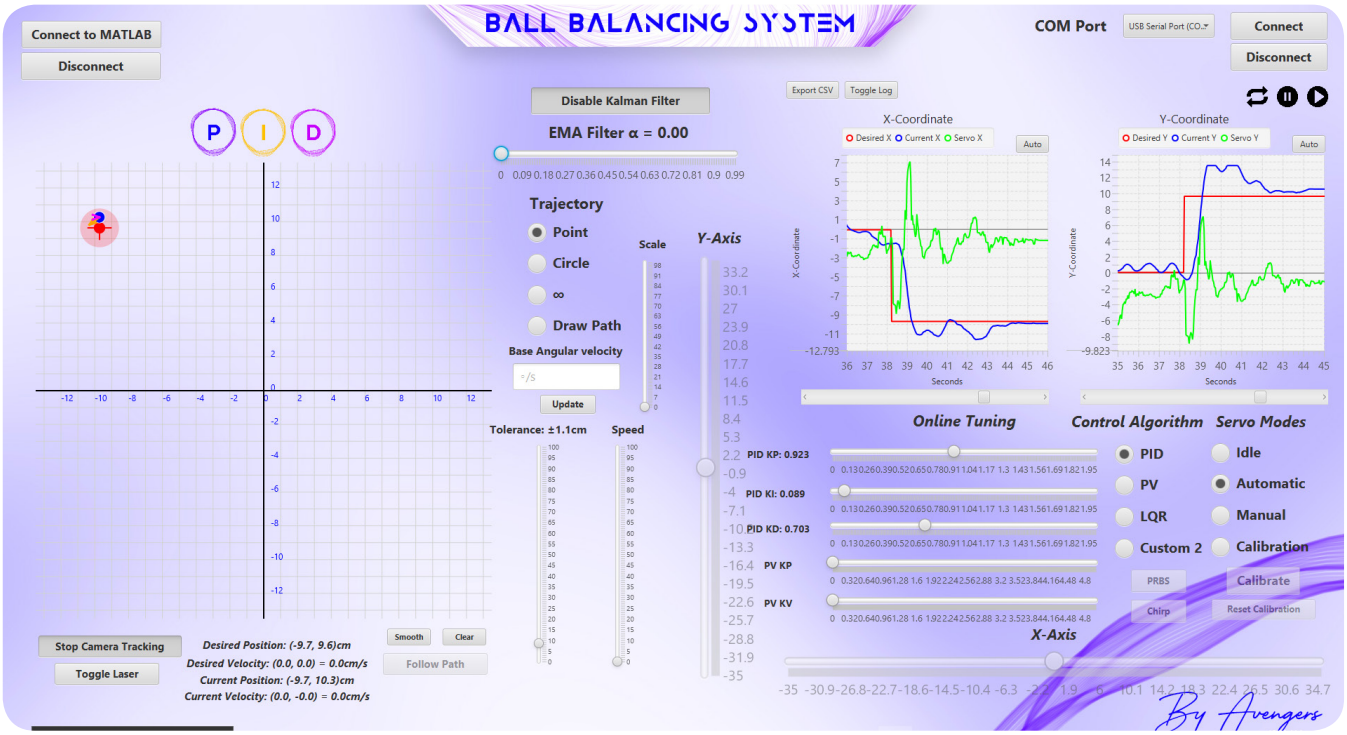
• Real-Time Monitoring: View live tracking of the ball's position on a coordinate grid.

• Trajectory Execution: Select predefined trajectories (e.g., circle or infinity) and input a desired angular velocity.

• Graphical Analysis: Visualize system input-output behavior along the X and Y axes through real-time plotted graphs.

• Data Logging: Export recorded data as CSV files for offline analysis and system performance evaluation.

• Online Tuning: Adjust control parameters (Kp, Ki, Kd, etc.) live during system operation for fine-tuning.



Java-Based Graphical User Interface (GUI)

2. Hardware Setup

The physical apparatus consists of a flat platform mounted on two servo motors arranged orthogonally. These motors control the platform's tilt along the X and Y axes. Key hardware components include:

• Servo Motors: Actuate platform angles based on control input.

• Webcam: Mounted above the platform to capture the ball's position in real-time.

• STM32F103C8T6 (Bluepill): A low-cost yet powerful microcontroller that serves as the control backbone of the system.

• Power Distribution Board

3. Image Processing and Ball Tracking

Using Python and OpenCV, the system continuously processes live video from the webcam to detect the ball's position:

• Image Acquisition: Captures video frames at runtime.

• Preprocessing: Filters and transforms frames to highlight the ball.

• Ball Detection: Locates the ball and extracts its X and Y coordinates.

• Tracking: Continuously monitors the movement to feed real-time data into the control loop.

Filtering techniques like the Exponential Moving Average (EMA) and optional Kalman Filter enhance position stability by minimizing noise.

4. Control Algorithms and STM32 Integration

At the heart of the system lies the STM32 microcontroller, which:

• Receives processed ball position data via Bluetooth from the PC.

• Applies the selected control algorithm to compute the necessary corrective action.

• Generates PWM signals to adjust servo motor angles.

Project Objectives

• Design a hardware platform using an STM32 microcontroller, webcam, and servo motors to balance a ball on a 2DOF surface.

• Develop real-time image processing algorithms to track the ball's X and Y coordinates.

• Implement and tune various control algorithms (PID, PV, LQR) for stable balancing.

• Create a Java-based GUI for full system control, trajectory selection, live plotting, and parameter tuning.

• Ensure seamless integration between software and hardware for responsive, accurate control.

Theoretical Model

• To derive the equations of motion for the ball-on-platform system, the following assumptions are considered:

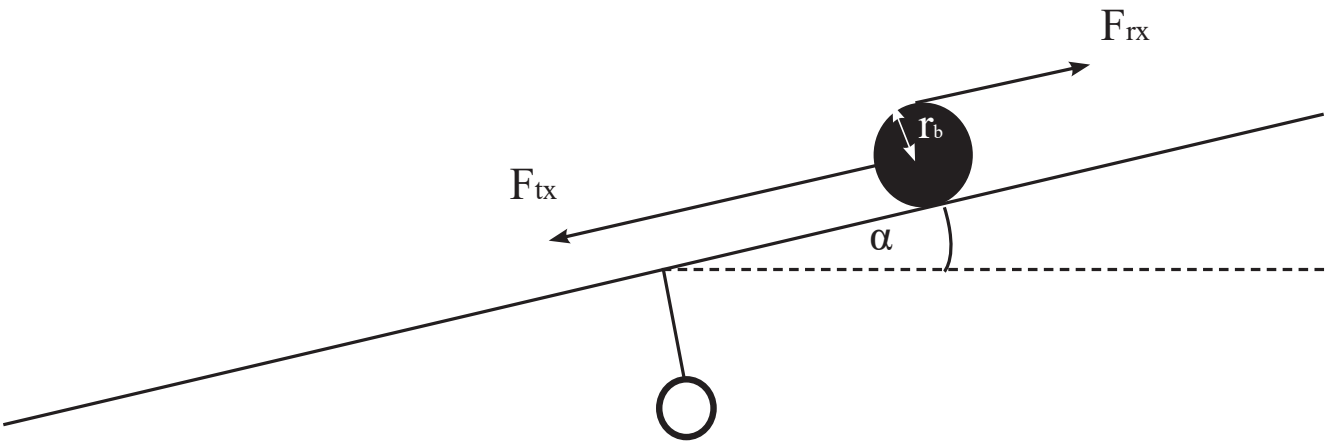
• The ball rolls without slipping.

• Frictional forces are neglected.

• The ball is perfectly spherical and has uniform mass distribution.

• No vertical displacement occurs between the ball and the platform.

Equations of Motion



Free-body diagram

Assuming there is no friction and damping in the system, the forces acting on the ball can be described as:

$$\sum F = F_{tx} - F_{rx} \qquad (1)$$

Where F_{tx} is the translational force acting on the ball due to gravity and inclined plane

$$F_{tx} = mg\sin(\alpha)$$

Frx is the force acting on the ball due to the ball's rotation. The torque generated is:

$$T = F_{xr} r_b = \frac{J_b}{r_b} \ddot{x}_b$$

Polar moment of inertia of a solid sphere and a thin shell hollow sphere are $J_b = \frac{2}{5} m_b r_b^2$ and $J_b = \frac{2}{3} m_b r_b^2$ respectively.

Thus, by substituting in equation (1)

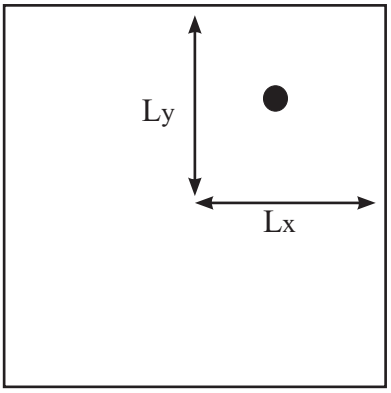
$$m_b\ddot{x}_b = m_b g\sin(\alpha) - \frac{J_b}{r^2{}_b}\ddot{x}_b \longrightarrow \boxed{\ddot{x}_b = \frac{m_b r^2{}_b g}{m_b r^2{}_b + J_b}\sin(\alpha)} \quad (2)$$

Applying the same method to the y-direction:

$$\boxed{\ddot{y}_b = \frac{m_b r^2{}_b g}{m_b r^2{}_b + J_b}\sin(\gamma)} \quad (3)$$

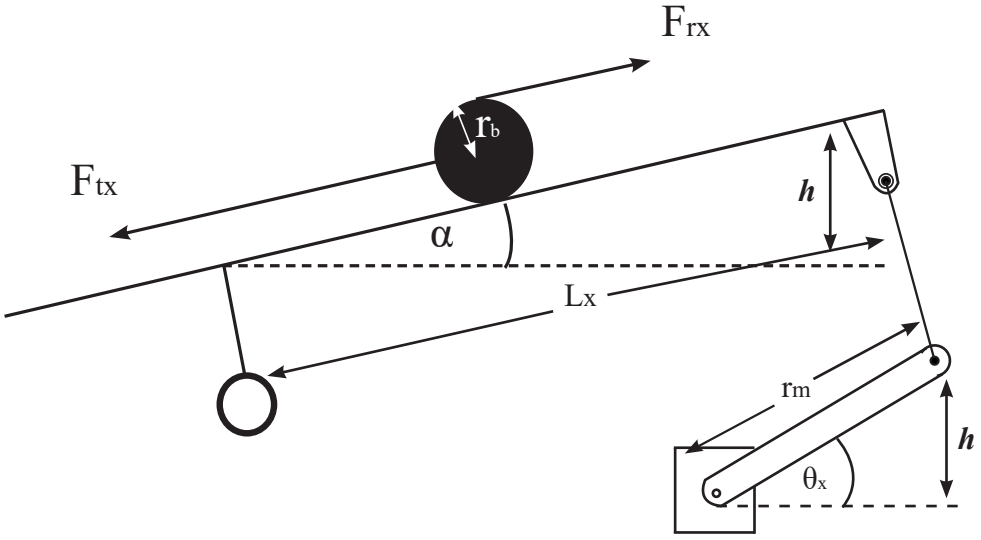
Where

- mb is the mass of the ball.
- g is the gravitational acceleration.
- Jb is the moment of inertia of ball.
- rb is the radius of ball.
- α is the plate tilt angle about x.
- γ is the plate tilt angle about y.
- Lx is the half length of the plate in x-direction.
- Ly is the half length of the plate in y-direction.
- r_m is length of servo link.
- θ_x & θ_y servo angles.



Top view of plate

The relationship between α and Servo title angle θ is as follows:



Relationship between Motor Angles and Plate Angles

$$\sin(\theta_x) r_m = \sin(\alpha_x) L_x \quad (4)$$

Also, for the y-direction:

$$\sin(\gamma) = \frac{r_m}{L_y}\sin(\theta_y) \quad (5)$$

We can substitute equations 4&5 in equations 2&3, we get the equation of motion of our system in terms of servo angles instead of plate title angles.

$$\ddot{x}_b = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_x} \sin(\theta_x) \quad (6) \qquad \ddot{y}_b = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_y} \sin(\theta_y) \quad (7)$$

Linearization around Operating Point:

To derive the system's transfer function, the nonlinear differential equations are linearized around the equilibrium point (x = 0, y = 0), assuming small-angle approximations:

$$\sin(\theta_x) \approx \theta_x \qquad \sin(\theta_y) \approx \theta_y$$

We can rewrite equations 6 & 7 as:

$$\ddot{x}_b = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_x} \theta_x \qquad \ddot{y}_b = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_y} \theta_y$$

Notably, the theoretical model is independent of both the ball's mass and radius.

Transfer Function of System Plant

Taking the Laplace transform we obtain the equation below:

$$\frac{X_b(s)}{\theta_x(s)} = \frac{k_x}{s^2} \qquad \frac{Y_b(s)}{\theta_y(s)} = \frac{k_y}{s^2}$$

$$k_x = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_x}$$

$$k_y = \frac{m_b r_b^2 r_m g}{(m_b r_b^2 + J_b) L_y}$$

Transfer Function of Servo Motors

Since the servomotors are equipped with built-in control circuits, there is no need to design separate servo controllers. Instead, the servomotor dynamics can be modeled as an ideal unit with a time delay, representing their ability to accurately and promptly execute the desired angular rotation.

$$G_M(s) = \frac{K_m}{\tau s + 1}$$

Where:

• K_m is the servo gain and time constant (τ) represents the time it takes for the system's response to reach approximately 63.2% of its final (steady-state) value after a step input.

$$K_m = \frac{Max\ anglular\ position - Min\ angular\ position}{Max\ input\ PWM - Min\ input\ PWM}$$

System Parameters

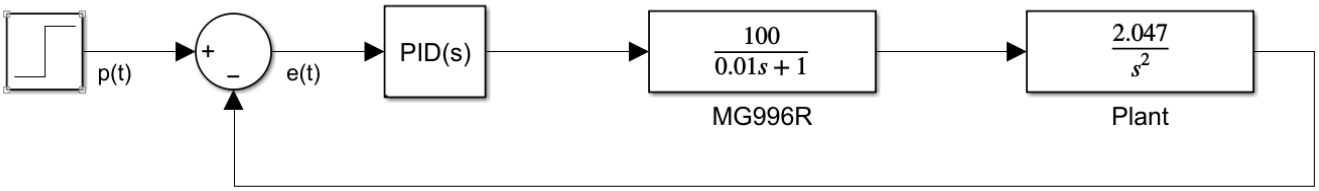
Parameters	Numerical Value
m_b	0.005 Kg
r_b	0.02m
r_m	0.047m
$L_x=L_y$	0.135m
g	$9.8m/s^2$
J_b	$1.333x10^{-6}$ $Kg.m^2$

Controller Design

Analyzing the open-loop transfer function of the plant (excluding the servo dynamics) reveals that the system is ***inherently unstable***, as it has two poles at the origin. This corresponds to a type 2 system with an open-loop transfer function of the form $\frac{k}{s^2}$.

As a result, the system exhibits ***no natural damping*** and will not settle without control action. To achieve closed-loop stability, at least one integrator (pole at the origin) must be compensated, and the root locus must be shifted into the left-half of the s-plane through proper controller design (e.g., adding zeros via PD or PID control, or state-space pole placement).

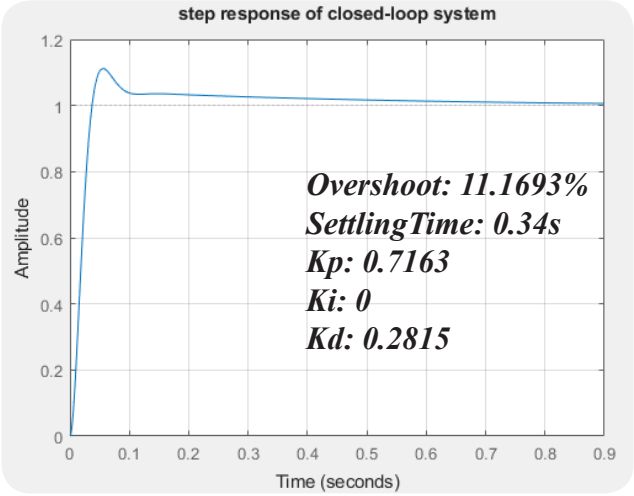
PID Controller Design

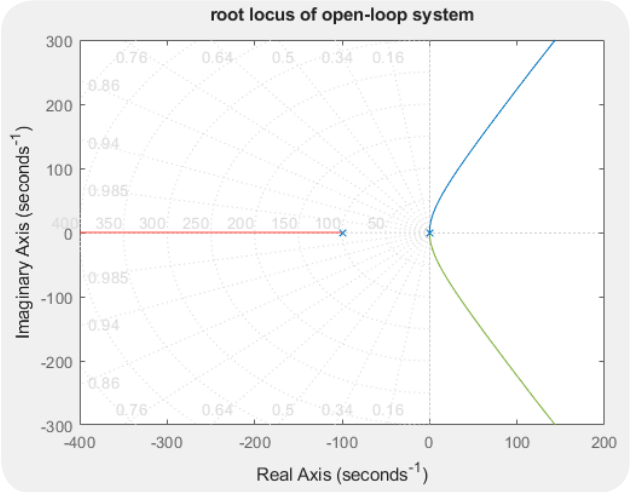


PID controller

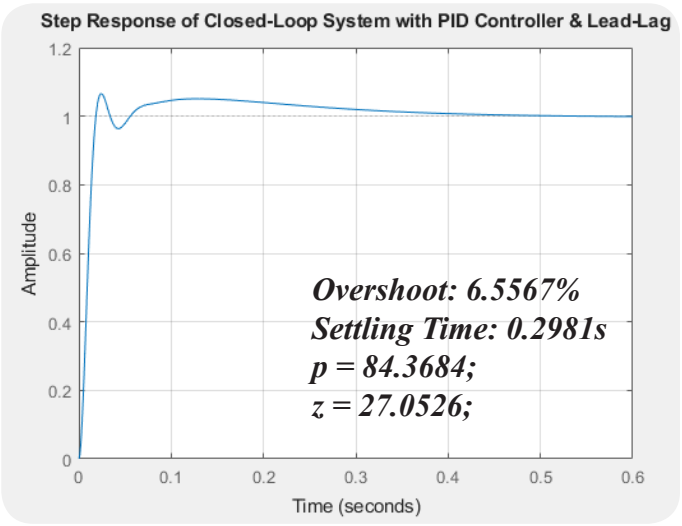
A PID controller can stabilize a double integrator system (with open-loop transfer function $\frac{k}{s^2}$) by appropriately tuning the proportional (Kp), integral (Ki), and derivative (Kd) gains. The derivative term (Kd) introduces a zero that adds phase lead, which contributes damping and helps shift the closed-loop poles into the left-half plane (LHP), ensuring asymptotic stability. However, the effectiveness of the PID controller depends heavily on proper tuning — poor tuning may lead to instability, oscillations, or sluggish system response.

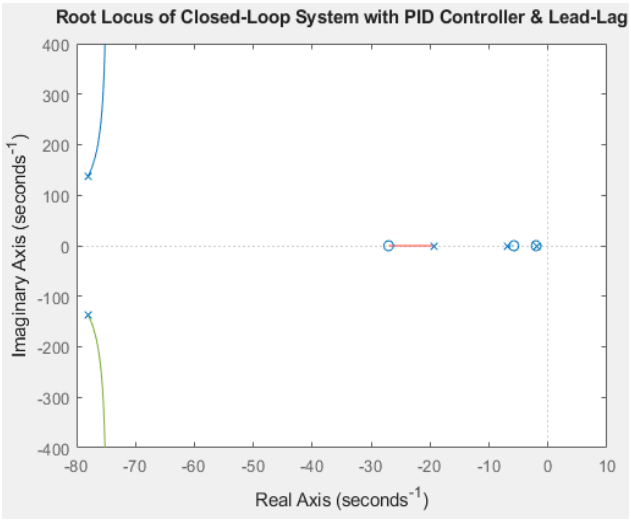
In theory, due to the system having a embedded integrator, the static error is self-regulated by the system itself. Thus, a integrating part could be redundant in the PID controller.





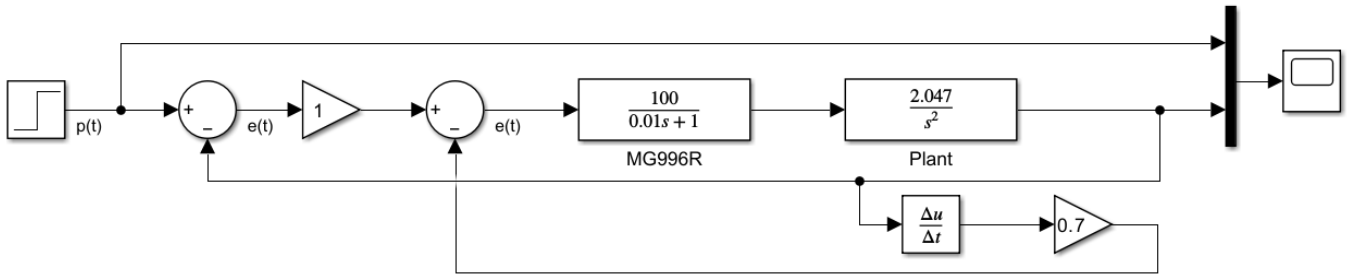
We suggest adding another Lead-Lag compensator, simulation gives far better results:



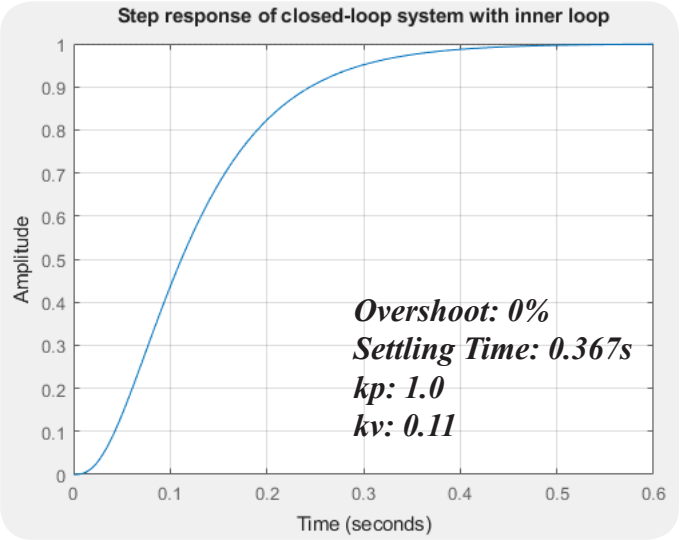


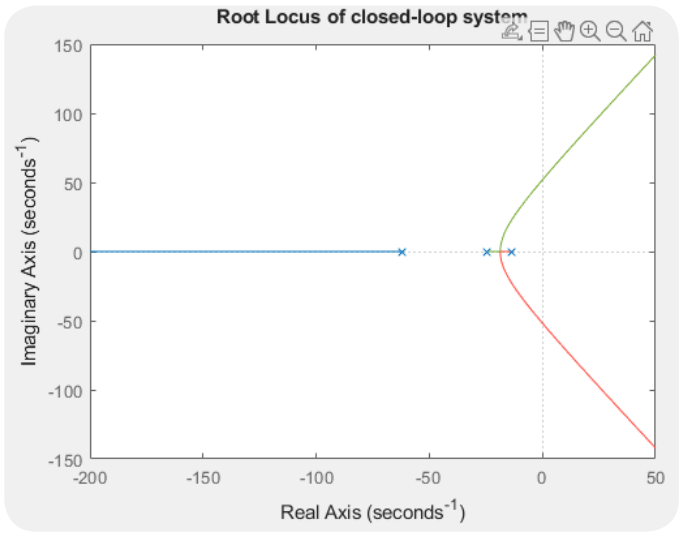
PD Controller Design

A PD controller is a commonly used servo position controller that enhances transient response and stability by introducing a zero (at -Kp/Kv) and adjusting the root locus to move the closed-loop poles to a more stable region. ***To avoid derivative kicks, the derivative term is often applied to the feedback signal rather than the error signal.*** In practical implementations, a low-pass filter is typically used to limit high-frequency noise amplification. However, in our system, since we already use a Kalman Filter in the image processing stage, which provides filtered data, an additional filter in the control loop is unnecessary. This setup offers flexibility in tuning the system's damping and response speed, but it is important to ensure that the system does not become overly sensitive to noise.



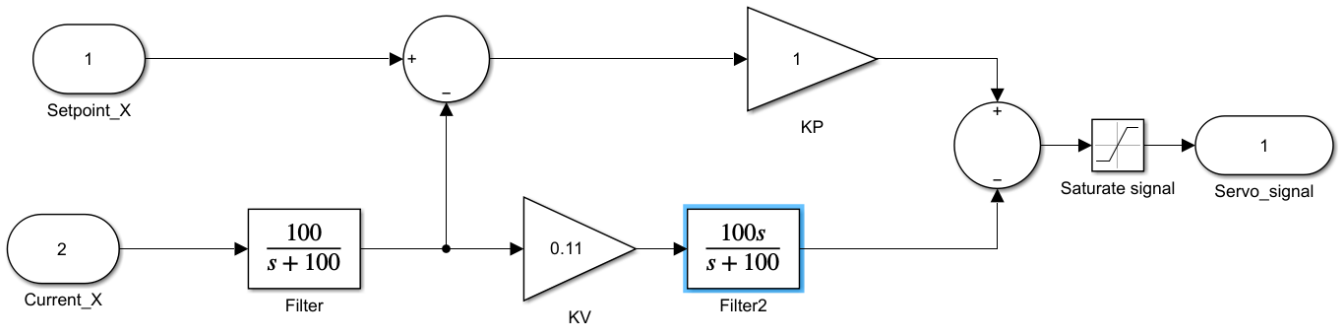
PD controller





Implementation

To implement the PD controller on our STM32 Bluepill and convert the Laplace domain to the discrete-time domain, we'll need to discretize both the controller and the low-pass filter, especially considering the cutoff frequency of 100rad/s (maintaining a reasonable noise rejection) for our low-pass filter (in case we disabled our kalman filter in the image processing stage).

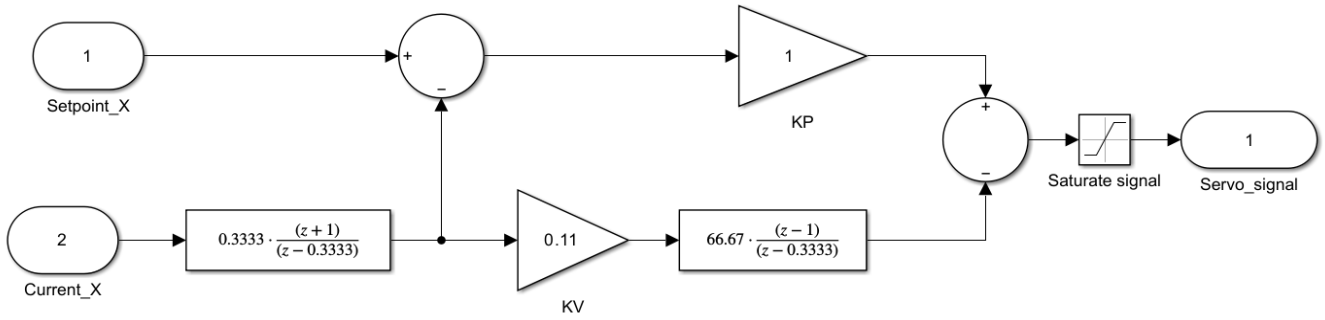


Block Diagram of the PD Controller in the Laplace Domain

Discretizing the Low pass filter using Bilinear Transform method (Tustin) in MATLAB:
Using sample time 0.01s (100Hz):

$$\frac{100}{s+100} \longrightarrow \boxed{\frac{0.3333\,(z+1)}{(z-0.3333)}}$$

$$\frac{100s}{s+100} \longrightarrow \boxed{\frac{66.67\,(z-1)}{(z-0.3333)}}$$



Block Diagram of the Discrete-Time PD Controller

Converting the transfer functions into difference equations:

$$H_{position}(z) = \frac{0.3333\,(z+1)}{(z-0.3333)} \quad \text{multiply both sides by (z-0.3333)}$$

$$y[z]\,(z-0.3333) = x[z]\,0.3333\,(z+1) \quad \text{expand terms}$$

$$zy[z] - 0.3333y[z] = 0.3333\,(zx[z] + x[z]) \quad \text{apply inverse z-transform}$$

$$y[n+1] = 0.3333\,(y[n] + x[n+1] + x[n])$$

Apply the causal form for real time implementation (shifting time):

$$\boxed{y[n] = 0.3333\,(y[n-1] + x[n] + x[n-1])}$$

$$\boxed{\begin{aligned} zx[z] &= x[n+1] \\ x[z] &= x[n] \\ z^{-1}x[z] &= x[n-1] \end{aligned}}$$

Inverse Z-transform

Applying same steps on the velocity filter:

$$y[z](z-0.3333) = x[z]\,66.67\,(z-1)$$
$$zy[z] = 0.3333y[z] + 66.67zx[z] - 66.67x[z]$$
$$y[n+1] = 0.3333y[n] + 66.67x[n+1] - 66.67x[n]$$
$$\boxed{y[n] = 0.3333y[n-1] + 66.67x[n] - 66.67x[n-1]}$$

Where:
• y[n] is the current output.
• y[n-1] is the previous output.
• x[n] is the current input.
• x[n-1] is the previous input.

Practical Implementation of PID Control for Ball Balancing System

An alternative approach to implementing the controller without discretizing the motion equations is to design a signal based on the current state variables:

$$Control\ Signal = K_P \cdot Error - K_V \cdot Velocity$$

In this formulation:
Kp is the proportional gain applied to the error between the desired and actual position of the ball and Kv is the derivative (or velocity) gain applied to the current velocity of the ball.

Since we use a Kalman filter to process the ball position obtained from the camera, we can extract a reliable estimate of the ball's velocity directly from the filter. This avoids the need for numerical differentiation, which is prone to noise.

However, in real-world implementation, ideal assumptions break down due to factors like static friction and system non-linearities.
For example:
• The ball may not begin to move until a certain threshold force is applied.
• There may be a steady-state error when tracking dynamic trajectories (e.g., a circular path), resulting in the ball following an offset trajectory.

To address these issues, we augment the controller by adding an integral term, resulting in a full PID controller:
• The integrator compensates for static friction and eliminates steady-state error.
• Note that we apply the derivative (velocity) gain on the velocity of the ball instead of the derivative of the error to avoid derivative kick.

Thus, the control law becomes:

$$Control\ Signal = K_P \cdot Error - K_V \cdot Velocity + K_I \cdot \int Error \cdot dt$$

Additional Smoothing via Filtering Output signal

To further improve system smoothness and reduce high-frequency oscillations or mechanical vibrations, we apply an Exponential Moving Average (EMA) filter to the controller's output:

$$Filtered\ Output(t) = \alpha \cdot Filtered\ Output(t-1) + (1-\alpha) \cdot Output(t)$$

• The filter coefficient α (between 0 and 1) determines the trade-off between responsiveness and smoothness.
• We expose α as a tunable parameter in the GUI, allowing real-time adjustments based on system behavior.

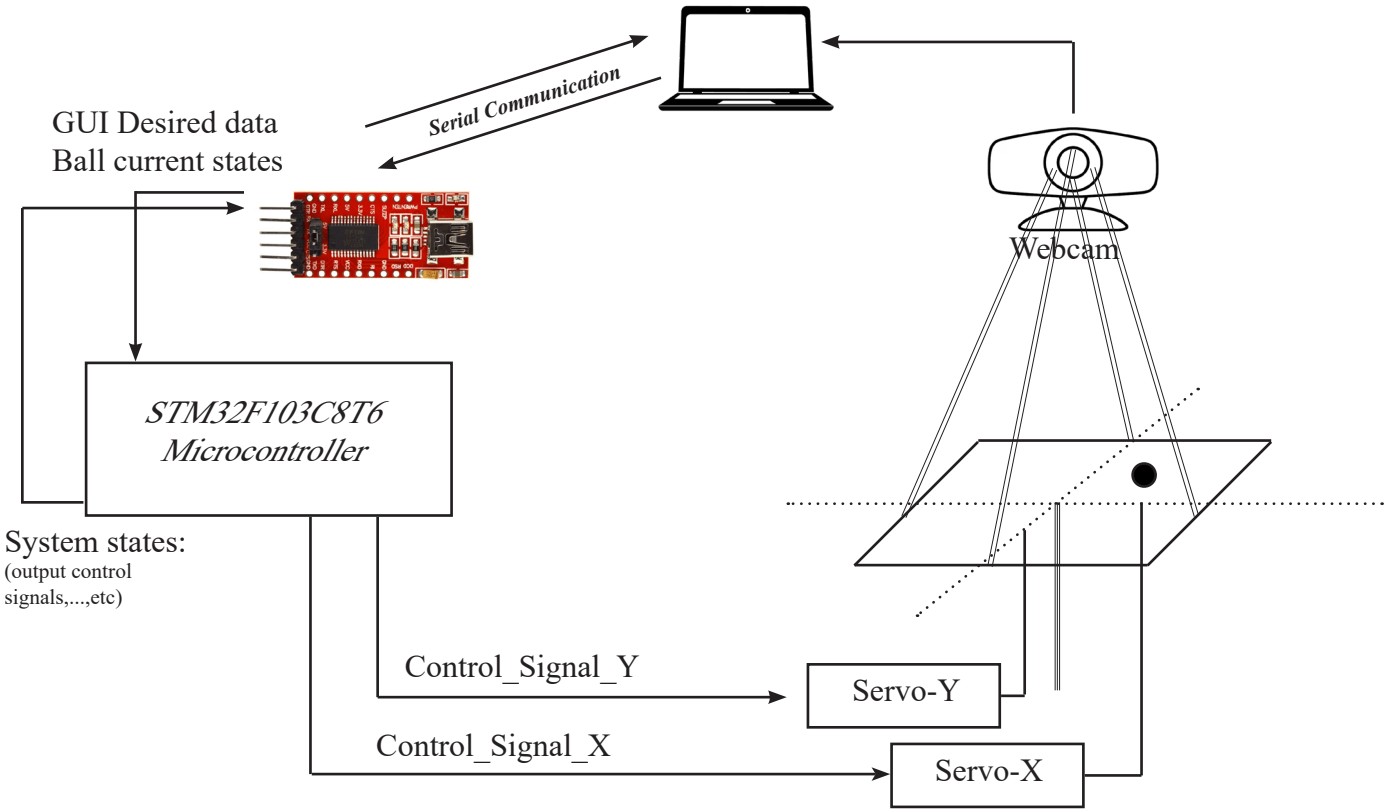
Since our system has multiple control objectives, we prioritized them as follows:
1-Preventing the ball from falling off the plate.
2-Reacting to external disturbances.
3-Controlling the ball position with different setpoints.
4-Trajectory tracking (e.g., circular path, figure-eight path, and custom-drawn paths via the GUI).

To meet these objectives, we implemented additional control strategies alongside the PID controller, such as the Linear Quadratic Regulator (LQR). However, a detailed explanation of our LQR implementation is beyond the scope of this report.

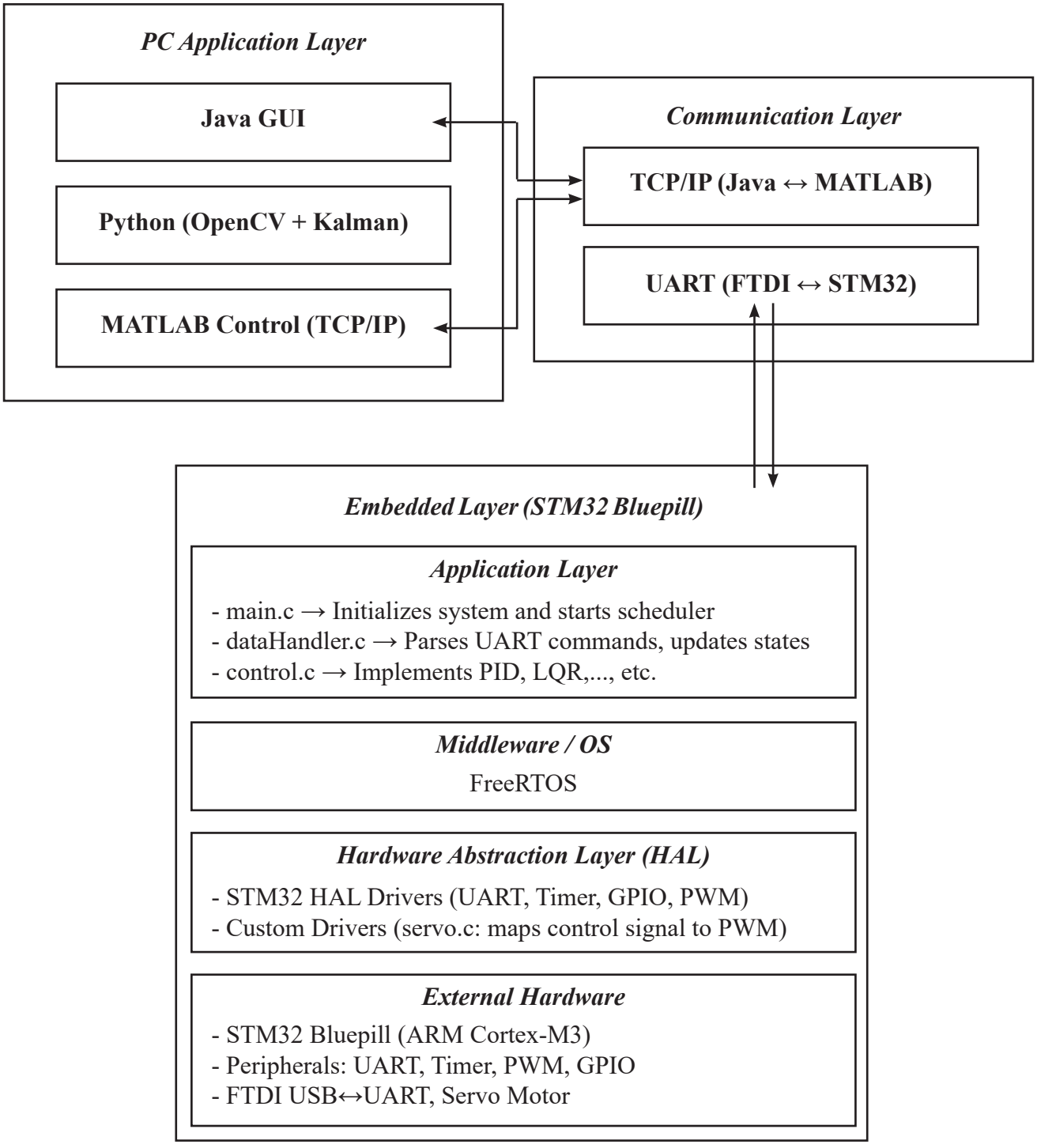
We also designed our system with future flexibility in mind. Specifically, we enabled the option for the control signal to be generated directly from a PC instead of being implemented in embedded C. In our GUI, we added a feature labeled "Connect to MATLAB". When enabled, this initializes a TCP/IP server between the Java-based GUI and MATLAB. We developed a MATLAB script that connects to this server, reads the current system states, and generates the control signal directly from a MATLAB or Simulink model, bypassing the need to reimplement the controller logic in C.

Moreover, the same communication interface can be used to connect a Python program to the system, allowing control signals to be generated from Python-based algorithms. This paves the way for future integration of Reinforcement Learning agents or other advanced control techniques developed in Python.

Functional Diagram



Layered Architecture Diagram Structure

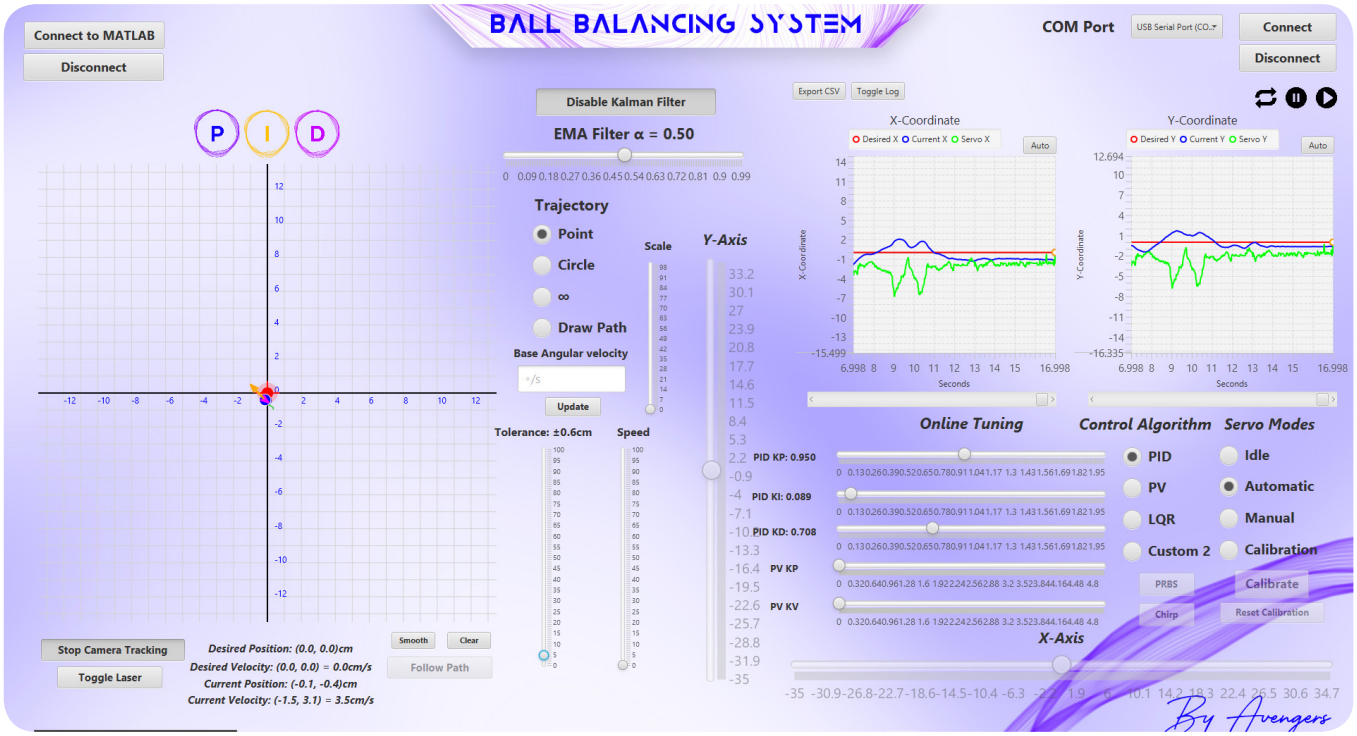


The embedded controller is complemented by PC-based tools that provide visualization, parameter tuning, trajectory planning, and optional control from higher-level algorithms.

1. Java Graphical User Interface (GUI)

Provides an interactive front-end for:

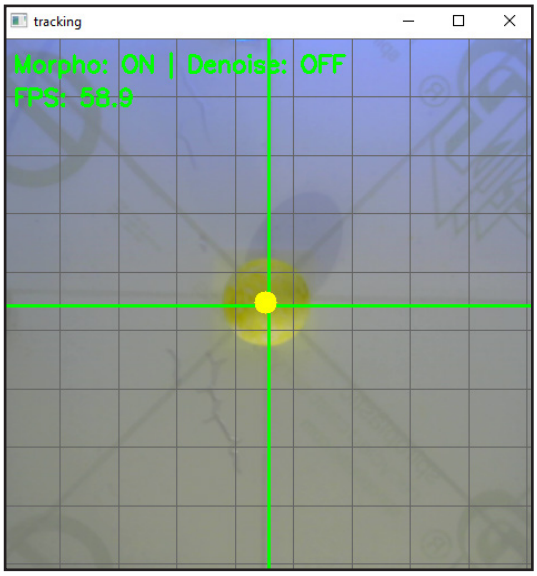
- Selection of operational modes.
- Selection of control modes.
- PID tuning and parameter entry.
- Custom trajectory drawing (e.g., circles, figure-eight, freeform paths).
- Communicates with the STM32 via UART through FTDI.
- Opens a TCP/IP socket for real-time communication with external software like MATLAB or Python.

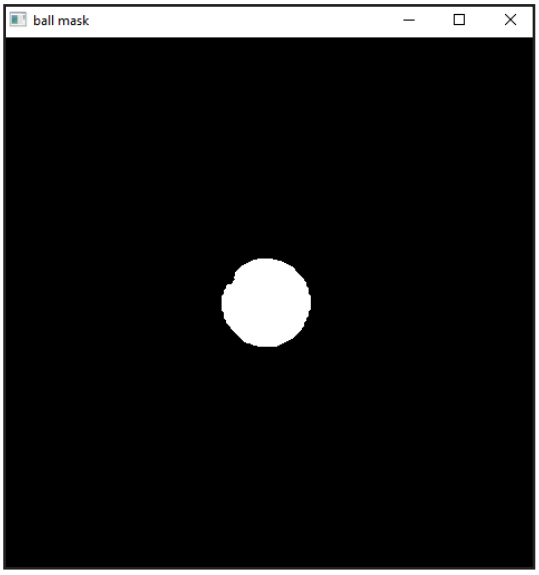


2. Python Program

Captures video using a USB webcam.

Uses computer vision and Image processing to track the ball's position and estimate velocity.





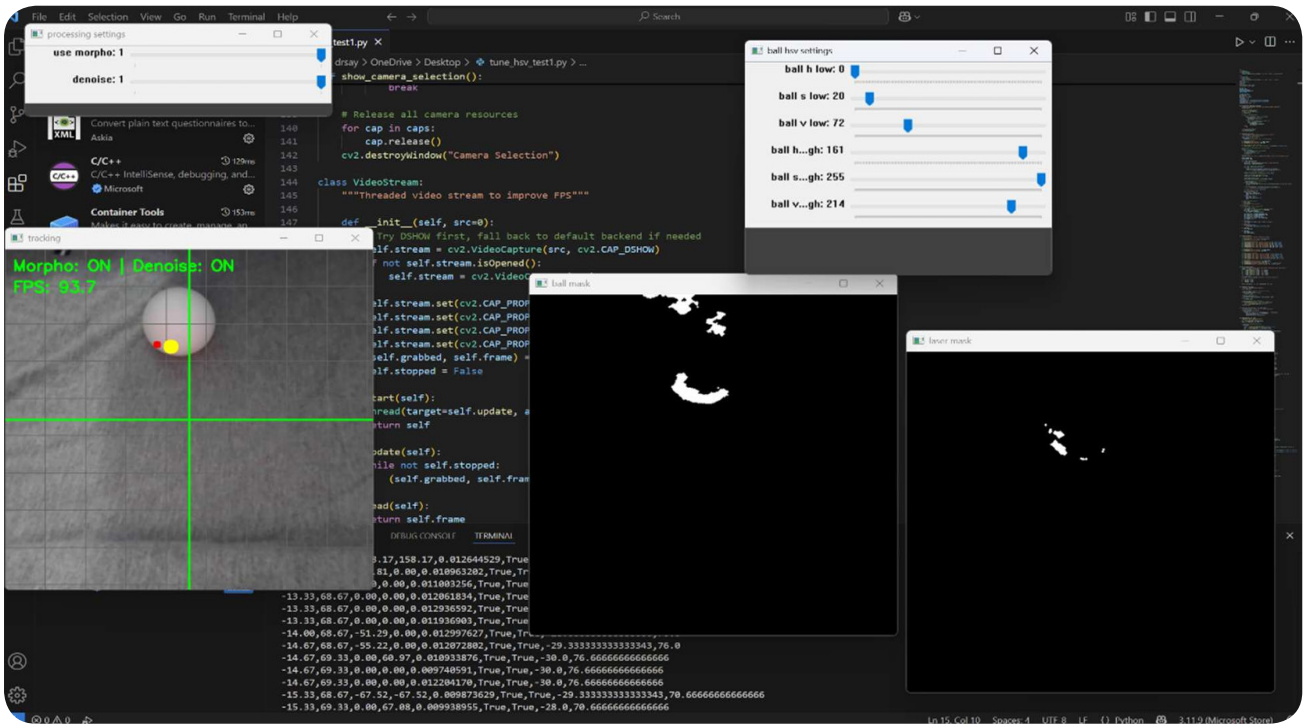
A Python-based image processing and tracking system was developed using OpenCV to enable accurate real-time control of the ball balancing platform. It captures and analyzes live videos to extract the positions of the ball and laser pointer, providing continuous feedback to the STM32 microcontroller for control computation. The system is composed of modular components— ranging from camera selection to data output—each optimized for real-time responsiveness and reliable performance in dynamic conditions.

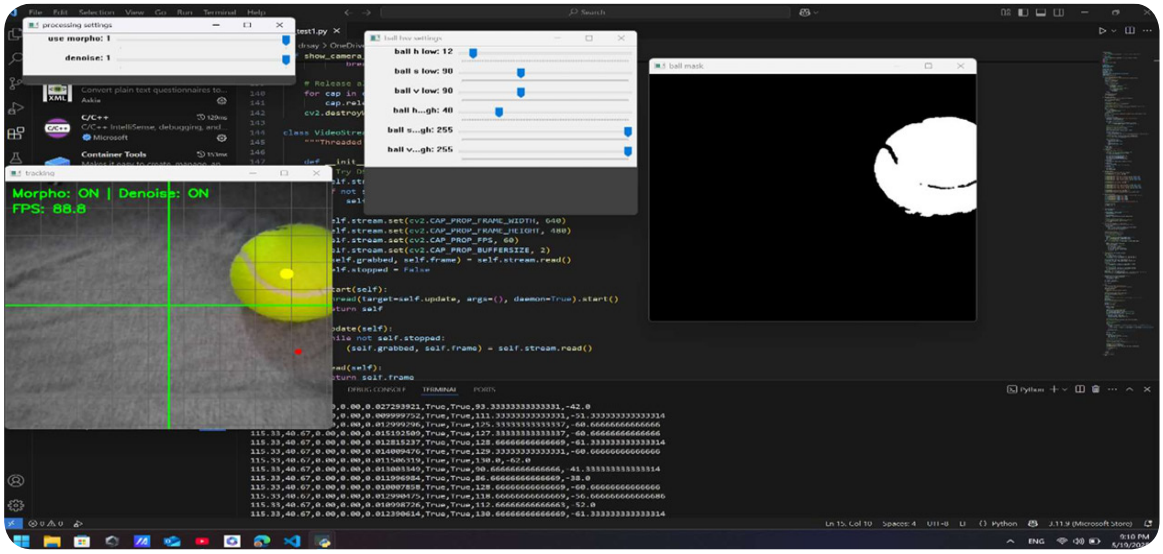
The Key Features

Camera Selection Interface: An intuitive GUI enables users to select from among connected cameras. Each camera offers a live preview, allowing users to choose the most suitable view of the platform. This feature enhances flexibility and ensures compatibility across different setups.

Threaded Video Capture: To reduce latency and improve frame acquisition rates, the system employs a multithreaded approach to video capture. By delegating frame acquisition to a separate thread, the program avoids bottlenecks and maintains tracking performance of up to 60 FPS.

HSV Tuning and Filtering: Object detection is based on HSV color space segmentation, which is more robust to lighting changes than BGR. The system provides independent, real-time trackbars to tune HSV thresholds for both the ball (usually yellow) and the laser pointer (typically red). This allows users to quickly adapt to varying lighting conditions and enhances detection accuracy.





The figure shows this HSV calibration showing yellow ball isolation vs. white ball rejection under identical lighting.

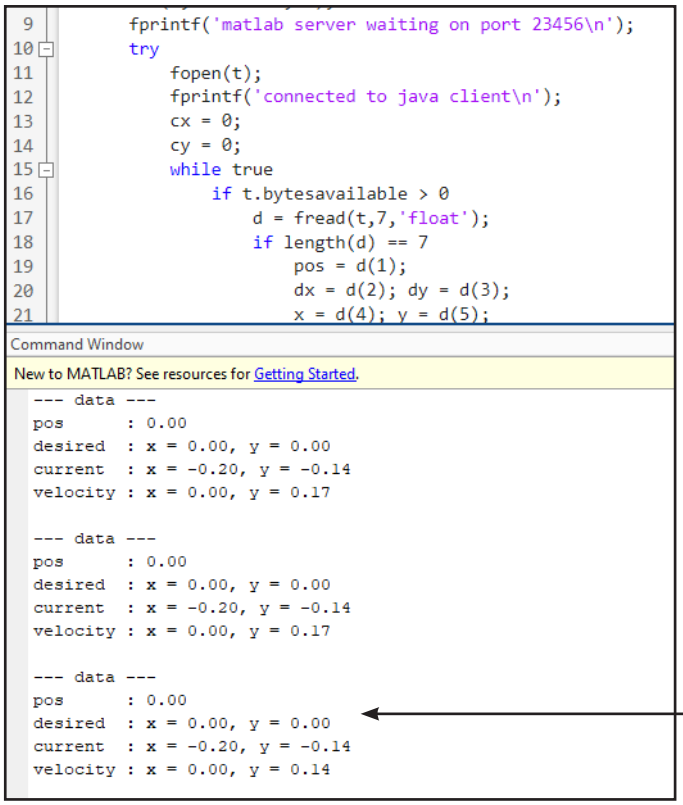
Image Preprocessing: To improve tracking stability, optional preprocessing techniques are applied to the video frames. These include Gaussian blurring to reduce noise and a moving average filter to smooth lighting fluctuations. Together, these steps ensure more consistent detection even under suboptimal conditions.

Object Detection and Position Estimation: Using contour analysis and morphological operations, the system identifies the centroid of the largest object matching the defined color filters. These centroid coordinates are then mapped to real-world positions (in millimeters) based on a calibrated 320mm grid. This conversion is crucial for generating accurate control signals for the balancing platform.

Velocity Estimation: To determine the ball's velocity, the system computes the positional change over time. A velocity buffer helps filter out noise and erratic movements, leading to smoother and more reliable velocity estimates, which are essential for stable control.

Visualization and Data Output: Each processed frame is overlaid with several visual aids, including: A coordinate grid aligned with the actual platform, Markers showing the detected positions of the ball and laser and Text overlays displaying FPS, filter status, and debug information. In addition to these visual outputs, the system prints comma-separated data containing position, velocity, and detection status. This data can be logged in real time for further analysis or troubleshooting. The tracking module communicates the ball’s coordinates to the STM32 microcontroller via Bluetooth. Once received, the STM32 uses this information to compute control signals that drive the platform's servo motors. Furthermore, the modular Python codebase is designed for extensibility, allowing for the future integration of control algorithms written in MATLAB through TCP/IP interfaces.

3. MATLAB / Simulink Integration (Optional)
MATLAB or Simulink can act as an external controller by interfacing through the Java GUI's TCP/IP socket.
Receives sensor feedback and sends calculated control signals to STM32.
Highly useful for academic experimentation and future improvements.



The architecture of the Ball Balancing Plate system is structured in clearly defined layers to ensure separation of modularity, scalability, and maintainability. This approach is aligned with best practices in embedded system design and facilitates system debugging, testing, and future expansion.

Application Layer
This layer contains the primary logic of the system, including communication, control, and actuation algorithms.
Core Files and Their Responsibilities
main.c:
• Initializes hardware and the RTOS environment.
• Launches the FreeRTOS scheduler.

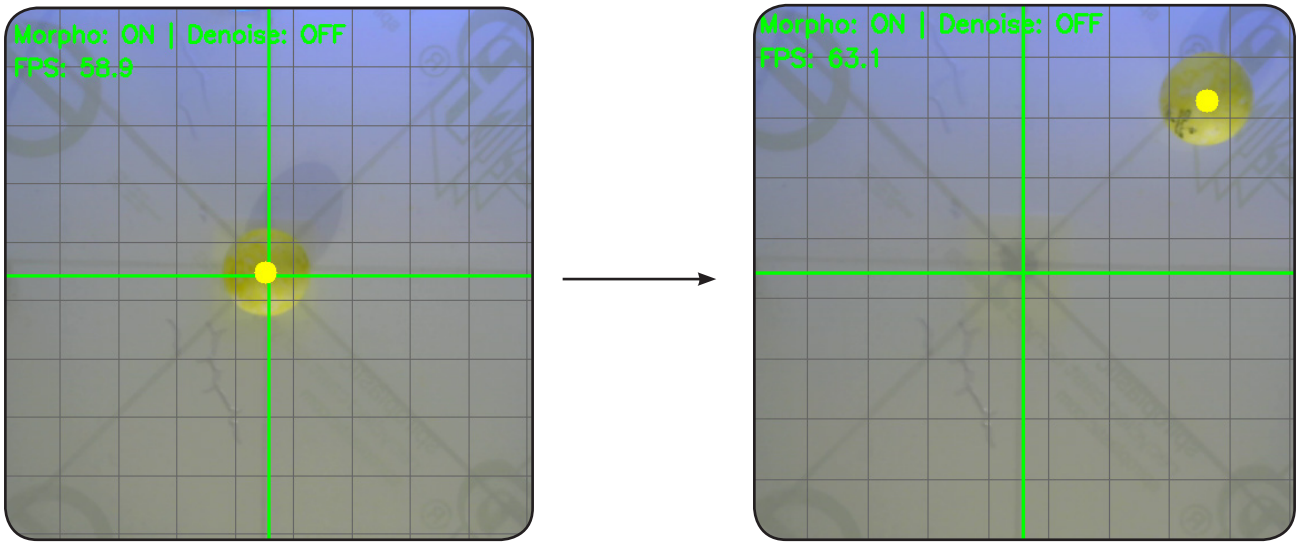
dataHandler.c:
• Parses incoming UART messages from the PC.
• Updates internal system states such as control parameters or operating mode.

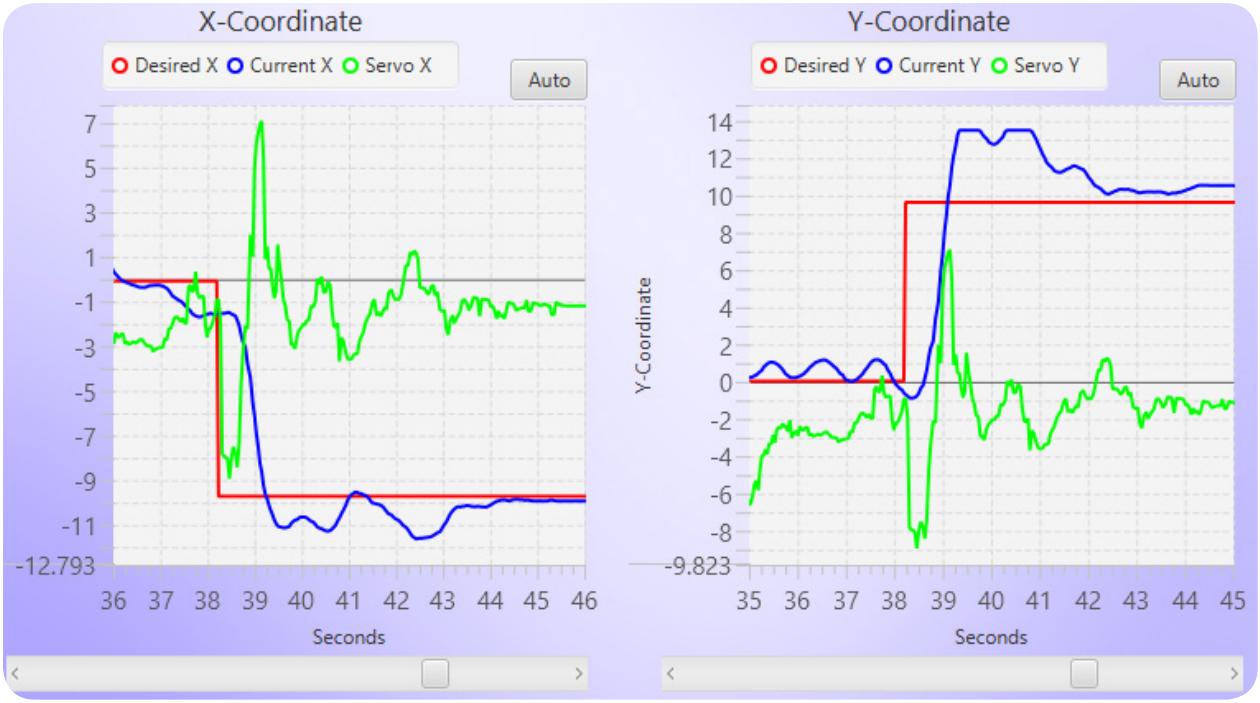
control.c:
• Implements control strategies (e.g., PID, PD, or LQR).
• Computes control signals based on real-time sensor input and target positions.
• Supports modular upgrades for advanced control schemes such as reinforcement learning.

servo.c:
• Converts the output of the control algorithms into precise servo commands.
• Ensures smooth and accurate physical actuation of the plate.

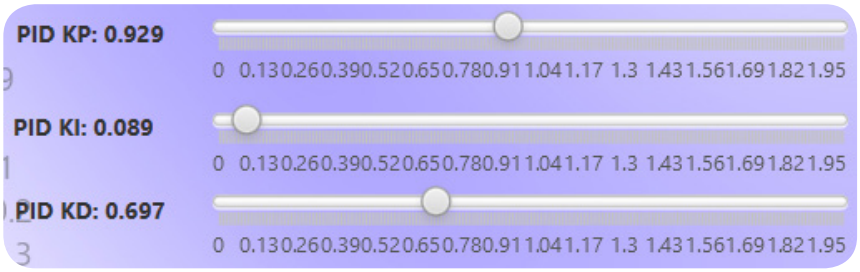
Control Response in Real-World Implementation

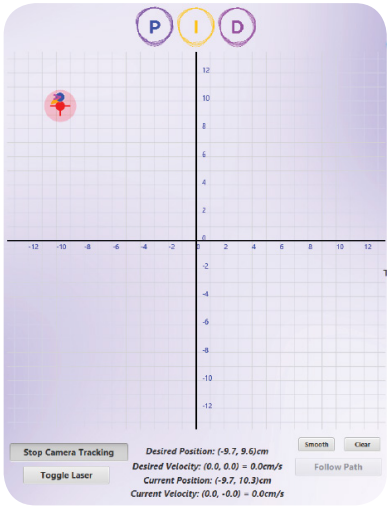
Step input response



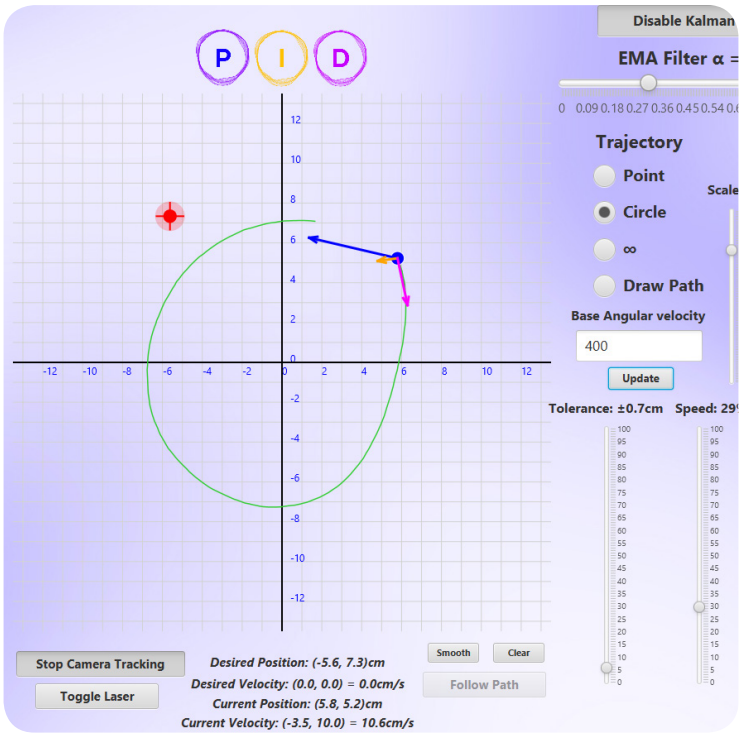


Note that the camera is rotated and the axes are rotated. setpoint(0,0) to setpoint(-9.7,9.6)

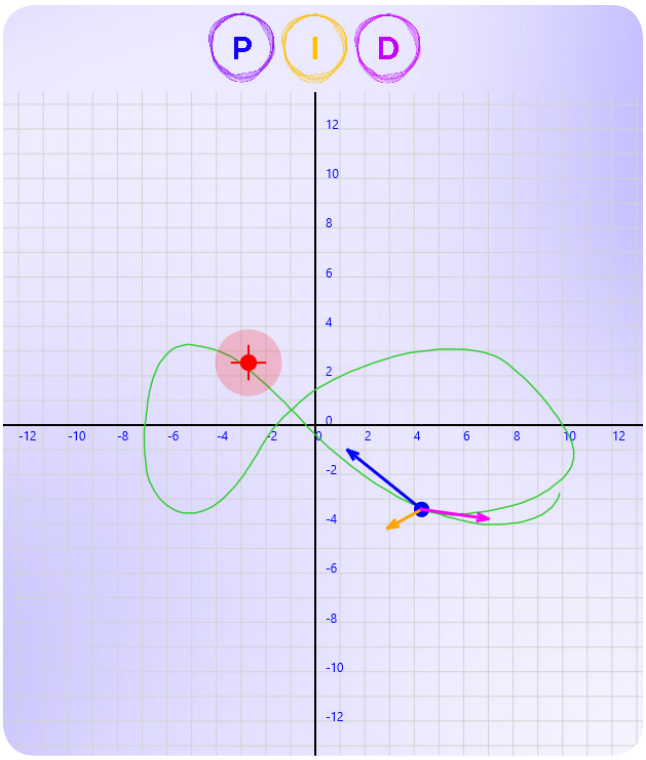




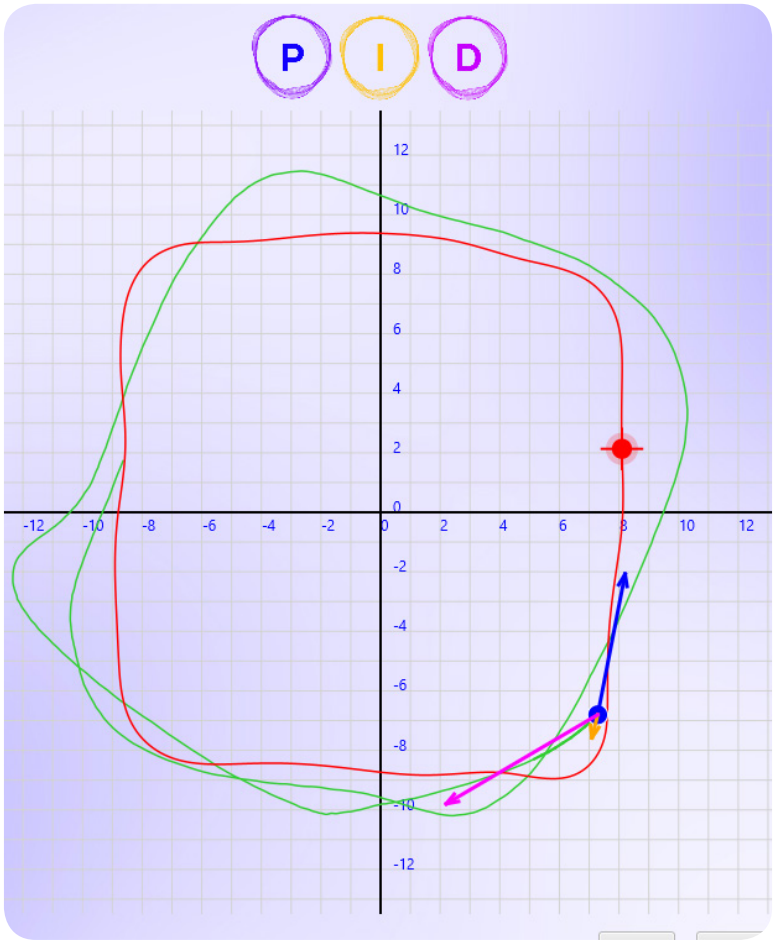
Circular Trajectory



8-figure trajectory



Custom drawn path



In the physical implementation of our system, the control response showed reliable performance in stabilizing the ball and tracking various trajectories. The system was able to maintain the ball at desired setpoints and follow paths such as circles and figure-eight patterns with acceptable accuracy.

However, some challenges emerged due to real-world factors such as mechanical friction, system latency, and slight delays in servo actuation. These effects, which were not present in the theoretical model, introduced minor steady-state errors and occasional deviations during motion.

To overcome these issues, we integrated an additional control term that helped eliminate steady-state errors, especially noticeable during continuous motion. For instance, when tracking a circular path, the ball's trajectory initially showed a slight offset from the intended path center. The added term corrected this behavior and improved overall accuracy.

To further enhance motion stability and reduce vibrations, we applied an exponential moving average (EMA) filter on the control signal. This smoothed out rapid changes while maintaining responsive performance. The smoothing factor (alpha) is adjustable via the GUI, allowing for flexibility based on the system's dynamics.

Mechanical Design

1. Motor Performance and Mounting

- Smooth, fluid motor movement with minimal friction.
- Secure, stable motor mounts to reduce vibrations.

2. Plate Rotation Mechanism

- Smooth, wobble-free rotation using low-friction bearings or bushings.

3. Arm and Structural Rigidity

- Rigid, lightweight servo arm to prevent flexing (e.g., aluminum or carbon fiber).

4. Connecting Rod Design

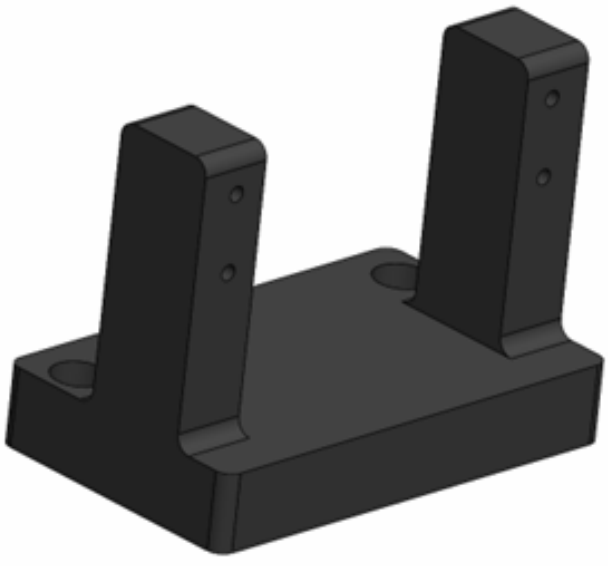
- Strong yet lightweight rod to efficiently transfer motion without bending.

5. Linkage and Joint Integrity

- Secure connections between the rod, plate, and arm for precise movement.

1. Motor Performance and Mounting

We selected the MG996R servo motor for this project due to its high torque output and smooth, precise motion, making it an ideal choice for our application.



To minimize energy losses caused by vibrations and ensure stable motor operation, we designed a dedicated servo mount.
The mount was 3D printed using PLA+ material, which offers excellent mechanical strength while maintaining ease of manufacturing. Additionally, we incorporated fillets at critical edges to reduce stress concentrations and improve the overall durability of the mount.

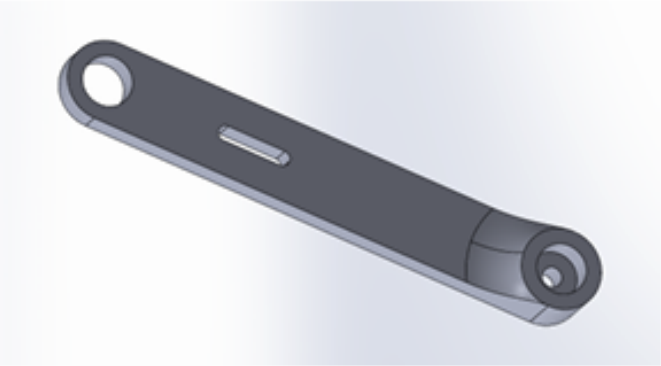
2. Plate Rotation Mechanism

This part of the assembly consists of three main parts and one variable part.

A. We used a ¾ in universal joint to ensure fluid movement with minimal friction.
B. Designed a modular mounting system that clips onto the universal joint from both sides, one is mounted to the base and the other to the plate.
C. Added a spacer to elevate the rotation system to compensate for the length of the connecting rod.

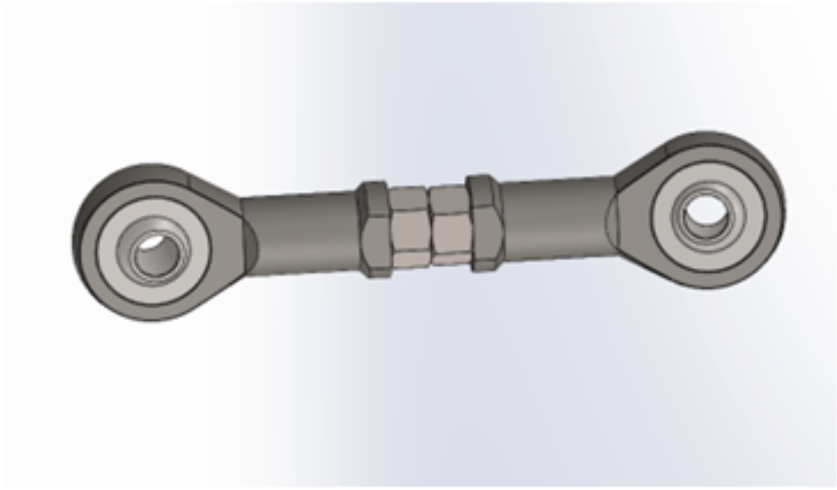
3. Arm and Structural Rigidity

An extended servo arm was designed to facilitate secure and reliable attachment to the connecting rod. The design includes a precision-aligned slot that allows the addition of a secondary screw, preventing any slippage during rotation. Additionally, a dedicated groove is incorporated to house the primary servo arm screw, ensuring proper placement and structural integrity.



4. Connecting Rod Design

We designed a connecting rod consisting of two BHS (Rod End) joints, enabling exceptionally smooth movement across all required axes. To minimize the overall weight of the rod, we selected the smallest available size on the market that met our performance requirements which was 5mm



5. Linkage and Joint Integrity

Because weight was a big concern, especially in the connecting rod, we used a 20mm 3D printed M5 threaded rod to connect the two BHS joints together and corresponding nuts for height adjustment. Similarly, the rod end is bolted using a 3D printed M5 screw and nut with the servo arm and a normal M3 screw at the top below the plate at which is a simple L-shaped link that connects the rod and the plate together.

Components:

• 2xMG996R Servo Motor
• FTDI usb to ttl ft232rl
• STM32F103C8T6 Blue pill
• USB Webcam
• Power Supply
• Buck Converter XL4015

Servo Motor (MG996R)

The platform's tilt mechanism is driven by two MG996R high-torque servo motors, providing precise two-axis control for ball stabilization. These industrial-grade servos were selected for their robust construction and reliable performance, with the following key characteristics:

System Implementation:
• Dual-Axis Configuration: Orthogonally mounted servos independently control X and Y tilt axes
• PWM Control: Driven by STM32-generated 50Hz PWM signals (500-2500μs pulse width)
• Mechanical Linkage: 47mm servo arms connected to platform with minimal backlash
Performance Specifications:
• Torque Output: 11kg·cm at 6V ensures adequate force for dynamic adjustments
• Angular Velocity: 0.15sec/60° enables rapid platform repositioning
• Positional Accuracy: ±1° resolution maintained through closed-loop feedback

Key Features:
• Metal Gear Transmission: Provides durability for continuous operation
• Wide Voltage Range: 4.8V-7.2V operation compatible with system power supply
• Integrated Potentiometer: Enables precise angle feedback without external sensors

Here the code in our STM32 that used for our System control :

1)main.c

```
#define RX_BUFFER_SIZE  256
//uint8_t rx_buffer[RX_BUFFER_SIZE];
//static uint8_t bufferIndex=0;
static volatile uint8_t rx_buffer[PACKET_SIZE + 3];  // Data + 2 start bytes + 1 checksum
static volatile uint16_t rx_index = 0;
static volatile bool sync_found = false;
static volatile uint8_t prev_byte = 0;
HAL_StatusTypeDef status;
DataHandler dataHandler;
MPU6050_HandleTypeDef mpu;
SemaphoreHandle_t dataHandler_Mutex = NULL;
QueueHandle_t uartRxQueue;
static volatile uint8_t rx_byte;
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_TIM1_Init();
  MX_USART2_UART_Init();
  Servo_Init();
  HAL_NVIC_SetPriority(USART2_IRQn, 4, 0); // Must be <= configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY (5)
  HAL_NVIC_EnableIRQ(USART2_IRQn);
  //HAL_Delay(1000);
  DataHandler_Init(&dataHandler);
  DataHandler_Reset(&dataHandler);
  Control_init(&dataHandler);
  dataHandler_Mutex = xSemaphoreCreateMutex();
  uartRxQueue = xQueueCreate(128, sizeof(uint8_t));
  HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
  xTaskCreate(UART_Task, "UART Task", 512, NULL, 2, NULL);
  xTaskCreate(updateController_Task, "Update Controller", 512, NULL, 1, NULL);
  vTaskStartScheduler();
```

```
void updateController_Task(void *parameters){
    const TickType_t period = pdMS_TO_TICKS(50);
    TickType_t xLastTick = xTaskGetTickCount();
        while (1) {
            xSemaphoreTake(dataHandler_Mutex, portMAX_DELAY);
            updateController_noUpdate();
            xSemaphoreGive(dataHandler_Mutex);
            vTaskDelayUntil(&xLastTick, period);

        }
}
```

```c
void UART_Task(void *parameters) {
    uint8_t current_byte;

    while(1) {
        if (xQueueReceive(uartRxQueue, &current_byte, portMAX_DELAY) == pdPASS) {
            if (xSemaphoreTake(dataHandler_Mutex, portMAX_DELAY) == pdTRUE) {
                if (!sync_found) {
                    if (prev_byte == START_BYTE_1 && current_byte == START_BYTE_2) {
                        sync_found = true;
                        rx_index = 0;
                        rx_buffer[rx_index++] = prev_byte;
                        rx_buffer[rx_index++] = current_byte;
                    }
                    prev_byte = current_byte;
                }
                else {
                    if (rx_index < sizeof(rx_buffer)) {
                        rx_buffer[rx_index++] = current_byte;
                    }

                    if (rx_index == sizeof(rx_buffer)) {
                        uint8_t checksum = 0;
                        for (uint16_t i = 2; i < PACKET_SIZE + 2; i++) {
                            checksum ^= rx_buffer[i];
                        }

                        if (checksum == rx_buffer[PACKET_SIZE + 2]) {
                            DataHandler_ProcessUARTData(&dataHandler, rx_buffer + 2, PACKET_SIZE);
                        }

                        sync_found = false;
                        rx_index = 0;
                    }
                }
                xSemaphoreGive(dataHandler_Mutex);
            }
        }
    }
}
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART2) {
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;
        xQueueSendFromISR(uartRxQueue, &rx_byte, &xHigherPriorityTaskWoken);
        HAL_UART_Receive_IT(huart, &rx_byte, 1);
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
}
```

```c
void updateController_noUpdate(void){
            static DataHandler* handler = &dataHandler;
            switch(DataHandler_GetOperationMode(handler)) {
                case MODE_IDLE:
                    Servo_X_SetAngle(0);
                    Servo_Y_SetAngle(0);
                    Control_Reset();break;
                case MODE_CALIBRATION:
                    Control_Reset();
                    switch(DataHandler_GetCalibrationMode(handler)){
                        case MODE_IDLE_CALIB:
                            Servo_X_Calib(DataHandler_GetCalibAngleX(handler));
                            Servo_Y_Calib(DataHandler_GetCalibAngleY(handler));
                            break;
                        case MODE_SAVE:
                            Servo_X_Save_Calib();
                            Servo_Y_Save_Calib();
                            break;
                        case MODE_RESET:
                            Servo_Reset_Calib();
                            break;}
                    break;
                case MODE_AUTOMATIC: {
                    switch (handler->trajectoryMode) {
                            case MODE_POINT:
                                Control_Loop();
                                break;
                            case MODE_CIRCLE:
                                generate_circle_points();
                                break;
                            case MODE_INFINITY:
                                generate_infinity_points();
                                break;
                            case MODE_PATH:
                                Control_Loop();
                                break;
                            default:
                                //Control_Loop();
                                break;}break;
                }
                case MODE_MANUAL:
                    Control_Reset();
                    Servo_X_SetAngle(DataHandler_GetManualAngleX(handler));
                    Servo_Y_SetAngle(DataHandler_GetManualAngleY(handler));
                    char debug_buf2[255];
                    snprintf(debug_buf2, sizeof(debug_buf2),
                             "{\"Servo_X\":%.2f,\"Servo_Y\":%.2f,\"desired_X\":%.2f,\"desired_Y\":%.2f}",
                             handler->manualAngleX, handler->manualAngleY,0.0f,0.0f);
                    UART_sendString(debug_buf2);
                    UART_sendString("\r\n");
                    memset(debug_buf2, 0, sizeof(debug_buf2));
                    break;
                case MODE_SAFE:
                    Control_Reset();
                    safeShutdownProcedure();
                    break;
                default:
                    break;}
```

```c
void safeShutdownProcedure(void){}
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART2) {
        uint8_t current_byte = rx_buffer[rx_index];
        if (!sync_found) {
            if (prev_byte == START_BYTE_1 && current_byte == START_BYTE_2) {
                sync_found = true;
                rx_index = 0;
                rx_buffer[rx_index++] = prev_byte;
                rx_buffer[rx_index++] = current_byte;
            }
            prev_byte = current_byte;
        }
        else {
            if (rx_index < sizeof(rx_buffer)) {
                rx_buffer[rx_index++] = current_byte;
            }

            if (rx_index == sizeof(rx_buffer)) {
                uint8_t checksum = 0;
                for (uint16_t i = 2; i < PACKET_SIZE + 2; i++) {
                    checksum ^= rx_buffer[i];
                }

                if (checksum == rx_buffer[PACKET_SIZE + 2]) {
                    DataHandler_ProcessUARTData(&dataHandler, rx_buffer + 2, PACKET_SIZE);
                }
                sync_found = false;
                rx_index = 0;
            }
        }
        HAL_UART_Receive_IT(huart, &rx_buffer[rx_index], 1);
    }
}
void UART_sendString(char *str) {HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);}
void UART_send_Char(char data) {HAL_UART_Transmit(&huart2, (uint8_t*)&data, 1, HAL_MAX_DELAY);}
void UART_receive_Data(char* data,uint16_t size){HAL_UART_Receive(&huart2, (uint8_t*)data, size, 1000);}
void UART_send_Data(char* data){HAL_UART_Transmit(&huart2, (uint8_t*)data, strlen(data), 1000);}
void UART_sendNumber(int32_t TxNumber) {
    if (TxNumber == 0) {
        UART_send_Char('0');
        return;
    }
    char buffer[12];
    int index = 0;
    if (TxNumber < 0) {UART_send_Char('-');TxNumber = -TxNumber;}
    while (TxNumber != 0) {buffer[index++] = (TxNumber % 10) + '0';TxNumber /= 10;}
    for (int i = index - 1; i >= 0; i--) {UART_send_Char(buffer[i]);}}
void SendPIDValues(DataHandler* handler) {
    char buffer[32];
    snprintf(buffer, sizeof(buffer), "PID_KP: %.4f\r\n", DataHandler_GetPID_KP(handler));UART_sendString(buffer);
    memset(buffer, 0, sizeof(buffer));
    snprintf(buffer, sizeof(buffer), "PID_KI: %.4f\r\n", DataHandler_GetPID_KI(handler));UART_sendString(buffer);
    memset(buffer, 0, sizeof(buffer));
    snprintf(buffer, sizeof(buffer), "PID_KD: %.4f\r\n", DataHandler_GetPID_KD(handler));UART_sendString(buffer);}
```

2)dataHandler.c:

```c
#include "data_handler.h"
#include <string.h>
#include "main.h"
static void parseDataPacket(DataHandler* handler, uint8_t* data);
static void handleError(DataHandler* handler, ErrorCode error);

void DataHandler_Init(DataHandler* handler) {
    if (handler == NULL) return;
    memset(handler, 0, sizeof(DataHandler));
    handler->operationMode = MODE_IDLE;
    handler->calibrationMode = MODE_IDLE;
    handler->controlMode = MODE_PID;
    handler->lastError = ERROR_NONE;
}
void DataHandler_ProcessUARTData(DataHandler* handler, uint8_t* data, uint16_t size) {
    if (handler == NULL || data == NULL || size != PACKET_SIZE) {
        handleError(handler, ERROR_PACKET_SIZE);
        return;}
    parseDataPacket(handler, data);
    handler->newDataAvailable = true;
}
static void parseDataPacket(DataHandler* handler, uint8_t* data) {
    uint8_t* ptr = data;
    // control Parameters
    memcpy(&handler->tolerance, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->desiredX, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->desiredY, ptr, sizeof(float)); ptr += sizeof(float);
    //velocity
    memcpy(&handler->current_velocity_X, ptr, sizeof(float)); ptr += sizeof(float); memcpy(&handler->current_velocity_Y, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->desired_velocity_X, ptr, sizeof(float)); ptr += sizeof(float);memcpy(&handler->desired_velocity_Y, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->calibAngleX, ptr, sizeof(float)); ptr += sizeof(float);// calibration Data
    memcpy(&handler->calibAngleY, ptr, sizeof(float)); ptr += sizeof(float);
    uint8_t opMode = *ptr++;// operation Modes
    uint8_t calMode = *ptr++;
    uint8_t ctrlMode = *ptr++;
    uint8_t trajMode = *ptr++;
    handler->operationMode = opMode;// validate modes
    handler->calibrationMode =calMode;
    handler->controlMode =ctrlMode;
    handler->trajectoryMode =trajMode;
    memcpy(&handler->manualAngleX, ptr, sizeof(float)); ptr += sizeof(float);// manual Control
    memcpy(&handler->manualAngleY, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->currentX, ptr, sizeof(float)); ptr += sizeof(float);// system State
    memcpy(&handler->currentY, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->PID_KP, ptr, sizeof(float)); ptr += sizeof(float);// pid parameters
    memcpy(&handler->PID_KI, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->PID_KD, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->PV_KP, ptr, sizeof(float)); ptr += sizeof(float);// pv parameters
    memcpy(&handler->PV_KV, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->ALPHA, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->trajectroy_scale, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->trajectroy_speed, ptr, sizeof(float)); ptr += sizeof(float);
    memcpy(&handler->trajectroy_base_speed, ptr, sizeof(float));ptr += sizeof(float);
    memcpy(&handler->dt, ptr, sizeof(float));ptr += sizeof(float);
    handler->ball_detected = *ptr;
}
```

```c
void DataHandler_Reset(DataHandler* handler) {
    if (handler == NULL) return;
    handler->tolerance = 0.0f;handler->desiredX = 0.0f;handler->desiredY = 0.0f;handler->current_velocity_X= 0.0f;handler->current_velocity_Y = 0.0f;
    handler->desired_velocity_X= 0.0f;handler->desired_velocity_Y = 0.0f;handler->calibAngleX = 0.0f;handler->calibAngleY = 0.0f;handler->operationMode = MODE_IDLE;
    handler->calibrationMode = MODE_IDLE;handler->controlMode = MODE_PID;handler->trajectoryMode = MODE_POINT;handler->manualAngleX = 0.0f;handler->manualAngleY = 0.0f;
    handler->currentX = 0.0f;handler->currentY = 0.0f;handler->PID_KP = 0.0f;handler->PID_KI = 0.0f;handler->PID_KD = 0.0f;handler->PV_KP = 0.0f;handler->PV_KV = 0.0f;
    handler->ALPHA = 0.0f;handler->dt = 0.0f;handler->lastError = ERROR_NONE;handler->newDataAvailable = false;// Status
}
void UART_sendDataHandler(DataHandler* data) {
    char buffer[512];
    int len = snprintf(buffer, sizeof(buffer),
        "tolerance=%.2f\n,desiredX=%.2f\n,desiredY=%.2f,\n"
        "current_velocity_X=%.2f\n,current_velocity_Y=%.2f,\n"
        "desired_velocity_X=%.2f\n,desired_velocity_Y=%.2f\n,"
        "calibAngleX=%.2f\n,calibAngleY=%.2f\n,"
        "operationMode=%d\n,calibrationMode=%d\n,controlMode=%d,\n"
        "manualAngleX=%.2f\n,manualAngleY=%.2f\n,"
        "currentX=%.2f\n,currentY=%.2f,\n"
        "PID_KP=%.2f\n,PID_KI=%.2f\n,PID_KD=%.2f,\n"
        "PV_KP=%.2f\n,PV_KV=%.2f\n,"
        "lastError=%d\n,newDataAvailable=%d\n\n",
        data->tolerance,data->desiredX,data->desiredY,data->current_velocity_X,data->current_velocity_Y,data->desired_velocity_X,
        data->desired_velocity_Y,data->calibAngleX,data->calibAngleY,(int)data->operationMode,(int)data->calibrationMode,(int)data->controlMode,
        data->manualAngleX,data->manualAngleY,data->currentX,data->currentY,data->PID_KP,data->PID_KI,data->PID_KD,data->PV_KP,data->PV_KV,
        (int)data->lastError,
        data->newDataAvailable ? 1 : 0
    );
    if (len > 0 && len < sizeof(buffer)) {UART_sendString(buffer);}
    else {UART_sendString("[ERROR] Data overflow\n");}}
static void handleError(DataHandler* handler, ErrorCode error) {
    if (handler == NULL) return;
    handler->lastError = error;}
float DataHandler_GetPID_KP(const DataHandler* handler) {return handler != NULL ? handler->PID_KP : 0.0f;}// Getter implementations (existing ones plus new ones)
float DataHandler_GetPID_KI(const DataHandler* handler) {return handler != NULL ? handler->PID_KI : 0.0f;}
float DataHandler_GetPID_KD(const DataHandler* handler) {return handler != NULL ? handler->PID_KD : 0.0f;}
float DataHandler_GetPV_KP(const DataHandler* handler) {return handler != NULL ? handler->PV_KP : 0.0f;}
float DataHandler_GetPV_KV(const DataHandler* handler) {return handler != NULL ? handler->PV_KV : 0.0f;}
CalibrationMode DataHandler_GetCalibrationMode(const DataHandler* handler) {return handler != NULL ? handler->calibrationMode : MODE_IDLE;}
ControlMode DataHandler_GetControlMode(const DataHandler* handler) {return handler != NULL ? handler->controlMode : MODE_PID;}
float DataHandler_GetTolerance(const DataHandler* handler) {return handler != NULL ? handler->tolerance : 0.0f;}
float DataHandler_GetDesiredX(const DataHandler* handler) {return handler != NULL ? handler->desiredX : 0.0f;}
float DataHandler_GetDesiredY(const DataHandler* handler) {return handler != NULL ? handler->desiredY : 0.0f;}
float DataHandler_GetCurrentVelocityX(const DataHandler* handler) {return handler != NULL ? handler->current_velocity_X : 0.0f;}
float DataHandler_GetCurrentVelocityY(const DataHandler* handler) {return handler != NULL ? handler->current_velocity_Y : 0.0f;}
float DataHandler_GetCalibAngleX(const DataHandler* handler) {return handler != NULL ? handler->calibAngleX : 0.0f;}
float DataHandler_GetCalibAngleY(const DataHandler* handler) {return handler != NULL ? handler->calibAngleY : 0.0f;}
OperationMode DataHandler_GetOperationMode(const DataHandler* handler) {return handler != NULL ? handler->operationMode : MODE_IDLE;}
float DataHandler_GetManualAngleX(const DataHandler* handler) {return handler != NULL ? handler->manualAngleX : 0.0f;}
float DataHandler_GetManualAngleY(const DataHandler* handler) {return handler != NULL ? handler->manualAngleY : 0.0f;}
float DataHandler_GetCurrentX(const DataHandler* handler) {return handler != NULL ? handler->currentX : 0.0f;}
float DataHandler_GetCurrentY(const DataHandler* handler) {return handler != NULL ? handler->currentY : 0.0f;}
ErrorCode DataHandler_GetLastError(const DataHandler* handler) {return handler != NULL ? handler->lastError : ERROR_NONE;}
bool DataHandler_IsNewDataAvailable(const DataHandler* handler) {return handler != NULL ? handler->newDataAvailable : false;}
void DataHandler_ClearNewDataFlag(DataHandler* handler) {if (handler != NULL) {handler->newDataAvailable = false;}}
float DataHandler_GetDesiredVelocityX(const DataHandler* handler){return handler != NULL ? handler->desired_velocity_X : false;}
float DataHandler_GetDesiredVelocityY(const DataHandler* handler){return handler != NULL ? handler->desired_velocity_Y : false;}
float DataHandler_GetALPHA(const DataHandler* handler){return handler != NULL ? handler->ALPHA: false;}
TrajectoryMode DataHandler_GetTrajectoryMode(const DataHandler* handler){return handler != NULL ? handler->trajectoryMode : false;}
```

3)servo.c:

```c
#include "servo.h"
#include "main.h"
#include "MPU6050.h"
#define FLASH_CALIBRATION_ADDRESS  0x0800F800

TIM_HandleTypeDef htim2;char debug_buf[64];
static float X_REF=-12.8f;static float Y_REF=-21.4f;static float current_x=0;static float current_y=0;static float pitch=0;static float roll=0;
void Servo_Init(void) {
    TIM_OC_InitTypeDef sConfigOC = {0};
    __HAL_RCC_TIM2_CLK_ENABLE();
    htim2.Instance = TIM2;htim2.Init.Prescaler = 63;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;htim2.Init.Period = 19999;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    HAL_TIM_PWM_Init(&htim2);
    sConfigOC.OCMode = TIM_OCMODE_PWM1;sConfigOC.Pulse = 1500;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1);
    HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
    X_REF=5.4f;Y_REF=-5.4f;//Flash_Read_Calibration();
    Servo_X_SetAngle(0);Servo_Y_SetAngle(0);}
void Servo_X_SetAngle(float angle) {
    if (angle < SERVO_MIN_ANGLE) angle =SERVO_MIN_ANGLE ;
    if (angle > SERVO_MAX_ANGLE ) angle = SERVO_MAX_ANGLE ;//MPU6050_CalculatePitchRoll(&pitch, &roll);
    angle+=X_REF+90+pitch;
    uint32_t pulse = 500 + (angle / 180.0) * 2000;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, pulse);}
void Servo_Y_SetAngle(float angle) {
    if (angle < SERVO_MIN_ANGLE) angle =SERVO_MIN_ANGLE ;
    if (angle > SERVO_MAX_ANGLE ) angle = SERVO_MAX_ANGLE ;
    //MPU6050_CalculatePitchRoll(&pitch, &roll);
    angle+=Y_REF+90+roll;
    uint32_t pulse = 500 + (angle / 180.0) * 2000;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, pulse);}
void Servo_X_Calib(float angle) {
    if (angle < SERVO_MIN_CALIB_ANGLE) angle =SERVO_MIN_CALIB_ANGLE ;
    if (angle > SERVO_MAX_CALIB_ANGLE ) angle = SERVO_MAX_CALIB_ANGLE ;
    current_x=angle+X_REF;
    angle=current_x+90;
    uint32_t pulse = 500 + (angle / 180.0) * 2000;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, pulse);//    sprintf(debug_buf, "XC:%.2f,P:%lu\r\n", angle-90, pulse);
//    UART_sendString(debug_buf);
}
void Servo_Y_Calib(float angle) {
    if (angle < SERVO_MIN_CALIB_ANGLE) angle =SERVO_MIN_CALIB_ANGLE ;
    if (angle > SERVO_MAX_CALIB_ANGLE ) angle = SERVO_MAX_CALIB_ANGLE ;
    current_y=angle+Y_REF;
    angle=current_y+90;
    uint32_t pulse = 500 + (angle / 180.0) * 2000;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, pulse);//    sprintf(debug_buf, "YC:%.2f,P:%lu\r\n", angle-90, pulse);//    UART_sendString(debug_buf);
}
void Servo_X_Save_Calib(void){X_REF=current_x;}//Flash_Write_Calibration();
void Servo_Y_Save_Calib(void){Y_REF=current_y;}//Flash_Write_Calibration();
void Servo_Reset_Calib(void){X_REF=0;Y_REF=0;current_x=0;current_y=0;} //Flash_Write_Calibration();
```

4)control.c:

```c
#include "servo.h"
#include "control.h"
#include "main.h"
#include <stdio.h>
#define FLASH_CONTROL_ADDRESS          0x0800FC00
#define FILTER_COEFF_A1 0.9802f
#define FILTER_COEFF_B1 0.0198f
#define OUTPUT_GAIN 20.0f
#define PWM_MIN 0.5
#define PWM_MAX 2.5
#define N 8
static float KP=0.6f;static float KI=0.0001f;static float KD=0.02f;
static float PV_KP=0.75755102f;static float PV_KV=0.09081633f;static float ALPHA=0.5;
PIDController PID_X;PIDController PID_Y;DataHandler* handler;
float desired_X=0.0f;float desired_Y=0.0f;float dt=0.0f;
TrajectoryMode trajectoryMode=MODE_POINT;
float current_X=0.0f;float current_Y=0.0f;float tolerance=1.0f;
volatile uint32_t previousTime = 0;volatile uint32_t currentTime = 0;
char debug_buf2[256];float current_state[4];float desired_state[4];float u[2];
LQR_Controller ctrl;float K[2][4] = {{10.0000, 5.4772, 0.0000, 0.0000},{0.0000, 0.0000, 10.0000, 5.4772}};
float A[4][4] = {{0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 0.0, 0.0, 0.0}};
float B[4][2] = {{0.0, 0.0}, {1.0, 0.0},{0.0, 0.0},{0.0, 1.0}};

void Control_init(DataHandler*  dataHandler){
    if(dataHandler!=NULL){handler=dataHandler;}
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;//Flash_Read_Control_Parameters();
    Control_PID_Init(&PID_X);Control_PID_Init(&PID_Y);Control_PV_Init();LQR_Init(&ctrl);
    currentTime=DWT->CYCCNT;
    UpdateTime();}

void Control_PID_Init(PIDController* pid) {
    *pid = (PIDController){
        .kp = KP,
        .ki = KI,
        .kd = KD,
        .integral = 0.0f,
        .tolerance = 0.0f,
        .desired_V = 0.0f,
        .current_V = 0.0f,
        .prev_output = 0.0f,
        .prev_measurement = 0,
        .prev_output = 0,
        .prev_time = HAL_GetTick(),
        .delta_pos={0,0,0,0,0,0,0,0},
        .filtered_delta_pos= 0.0f
    };
}
PV_ControllerStates ctrl_x;PV_ControllerStates ctrl_y;
```

```
void Control_PV_Init(void) {
    ctrl_x.prev_pos_filtered = 0.0f;
    ctrl_x.prev_vel_filtered = 0.0f;
    ctrl_x.prev_vel_input = 0.0f;
    ctrl_x.prev_pos = 0.0f;
    ctrl_x.prev_output = 0.0f;

    ctrl_y.prev_pos_filtered = 0.0f;
    ctrl_y.prev_vel_filtered = 0.0f;
    ctrl_y.prev_vel_input = 0.0f;
    ctrl_y.prev_pos = 0.0f;
    ctrl_y.prev_output = 0.0f;
}
void UpdateTime() {
    previousTime = currentTime;
    currentTime = DWT->CYCCNT;
}
float GetDeltaTime() {return (currentTime - previousTime) / SystemCoreClock;}
void LQR_Init(LQR_Controller* ctrl) {
    // Initialize with your gains
    const float K_init[2][4] = {
        {5.6747f, 0.7304f, 0.0f, 0.0f},
        {0.0f, 0.0f, 5.6747f, 0.7304f}
    };
    const float Ki_init[2][2] = {
        {5.0f, 0.0f},
        {0.0f, 5.0f}
    };

    memcpy(ctrl->K, K_init, sizeof(K_init));
    memcpy(ctrl->Ki, Ki_init, sizeof(Ki_init));
    memset(ctrl->xi, 0, sizeof(ctrl->xi));
}
void LQR_Update(LQR_Controller* ctrl, const float x[4], const float r[2], float dt, float u[2]) {
    float y[2] = {x[0], x[2]};
    float e[2];

    e[0] = r[0] - y[0];
    e[1] = r[1] - y[1];

    ctrl->xi[0] += e[0] * dt;
    ctrl->xi[1] += e[1] * dt;

    ctrl->xi[0] = fmaxf(fminf(ctrl->xi[0], 2.0f), -2.0f);
    ctrl->xi[1] = fmaxf(fminf(ctrl->xi[1], 2.0f), -2.0f);

    u[0] = -(ctrl->K[0][0]*x[0] + ctrl->K[0][1]*x[1] +
            ctrl->K[0][2]*x[2] + ctrl->K[0][3]*x[3] +
            ctrl->Ki[0][0]*ctrl->xi[0] + ctrl->Ki[0][1]*ctrl->xi[1]);

    u[1] = -(ctrl->K[1][0]*x[0] + ctrl->K[1][1]*x[1] +
            ctrl->K[1][2]*x[2] + ctrl->K[1][3]*x[3] +
            ctrl->Ki[1][0]*ctrl->xi[0] + ctrl->Ki[1][1]*ctrl->xi[1]);
}
```

```
void Control_LQR_UpdateState(float x[4], const float u[2], float dt) {
    float x_dot[4] = {0};

    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            x_dot[i] += A[i][j] * x[j];
        }
        for (int j = 0; j < 2; j++) {
            x_dot[i] += B[i][j] * u[j];
        }
    }

    for (int i = 0; i < 4; i++) {
        x[i] += x_dot[i] * dt;
    }
}

void Control_LQR(const float current_state[4], const float desired_state[4], float control_output[2]) {
    float error[4];

    for (int i = 0; i < 4; i++) {
        error[i] = current_state[i] - desired_state[i];
    }

    control_output[0] = -(K[0][0]*error[0] + K[0][1]*error[1] +
                          K[0][2]*error[2] + K[0][3]*error[3]);

    control_output[1] = -(K[1][0]*error[0] + K[1][1]*error[1] +
                          K[1][2]*error[2] + K[1][3]*error[3]);
}

bool newsetpoint=false;
```

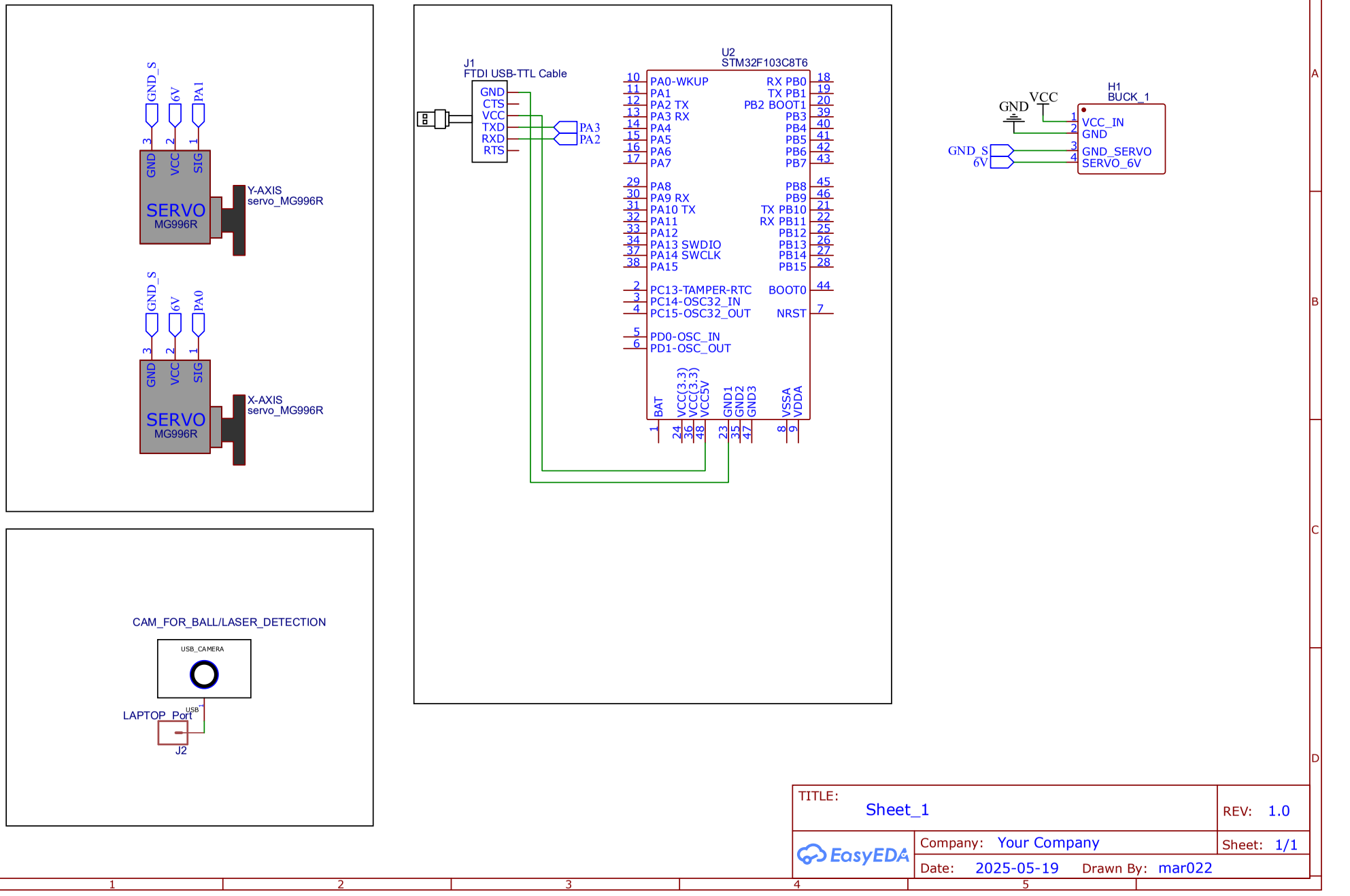
```c
#include <math.h>
void Control_PID2(PIDController* pid, float setpoint, float measurement, float* output) {
    float dt = 0.01f; // to be measured later after FTDI

    // === position prediction ===
//    float theta = (pid->prev_output / 180.0f) * 3.14f;
//    float accel = 2.047 * sin(theta);
//    pid->current_V = pid->current_V + accel * dt;
//    float predicted_position = measurement + pid->current_V * dt + 0.5f * accel * dt * dt;
    float error = setpoint - measurement;
    float abs_error = fabsf(error);
    // === adaptive gain ===
    float error_normalized = fminf(abs_error / 13.5f, 1.0f);
    float velocity_normalized = fminf(fabsf(pid->current_V) / 40.0f, 1.0f);

    float adaptive_kp = pid->kp * (0.3f + 0.7f * error_normalized);
    float adaptive_kd = pid->kd * (0.5f + 0.5f * velocity_normalized);
    float adaptive_ki = pid->ki * (0.3f + 0.7f * error_normalized);
    bool within_tolerance = (abs_error <= pid->tolerance);
    if (within_tolerance) {
        adaptive_kp *= 0.1f;
        adaptive_kd *= 0.5f;
        adaptive_ki = 0.0f;
        float prop = adaptive_kp * error;
        float derivative = -adaptive_kd * pid->current_V;
        pid->derivative=derivative;
        pid->prop=prop;
        pid->integral *= 0.99f;
        float output_raw = prop + pid->integral + derivative;
        float filter_alpha = 0.5f;
        *output = filter_alpha * pid->prev_output + (1.0f - filter_alpha) * output_raw;
        pid->prev_output = *output;
        return;
    }

    // === increase damping ===
    if (copysignf(1.0f, pid->current_V) != copysignf(1.0f, error)) {
        float damping_boost = 2.5f + 4.0f * expf(-fabsf(error) /0.03);
        adaptive_kd *= damping_boost;
    }

    // === braking Zone ===
    float brake_zone = 2.0f + 3.0f * (1.0f - expf(-fabsf(pid->current_V)/5.0f));
    if (fabsf(measurement - setpoint) < brake_zone &&
       fabsf(pid->current_V) > (5.0f*handler->trajectroy_speed)) {
        adaptive_kd *= 1.5f + velocity_normalized;
    }

    // === intergral I ===
    if((pid->current_V>1.0f) || abs_error>(10.0f*handler->trajectroy_scale)){
        pid->integral += adaptive_ki * error * dt;
    }
    pid->integral = fmaxf(fminf(pid->integral, 20.0f), -20.0f);

    // === pid output ===
    float prop = adaptive_kp * error;
    float derivative = -adaptive_kd * pid->current_V;
    float output_raw = prop + pid->integral + derivative;

    *output = ALPHA * pid->prev_output + (1.0f - ALPHA) * output_raw;
    pid->prev_output = *output;
}
```

```c
void Control_PID(PIDController* pid,float setpoint, float measurement, float* output){
    //UpdateTime();
    //dt = GetDeltaTime();
    if(handler->ball_detected ==0){
        pid->derivative=0;
        pid->integral=0;
        pid->prop=0;
        *output=0;
        return;
    }
    dt=handler->dt;
    float error= setpoint-measurement;

    float prop=pid->kp*error;
    pid->prop=prop;
    float derivative=(pid->kd) * (-(pid->current_V));
    if((newsetpoint && trajectoryMode==MODE_POINT) ){
        pid->integral=0;

    }
    if ((fabs(error) < tolerance)) {
//          pid->integral = 0.0f;
            derivative=0;
//          pid->prop=0;
//          *output=0;
//          return;
        }
        else{
            pid->integral+=(pid->ki)*(error)*dt;
        }
    float integral_limit = 13.5f;
    if (pid->integral > integral_limit) pid->integral = integral_limit;
    if (pid->integral < -integral_limit) pid->integral = -integral_limit;
    pid->derivative=derivative;


    pid->prev_error=error;

    float output_raw=prop+pid->integral+derivative;
    // EMA
    *output = ALPHA * pid->prev_output + (1 - ALPHA) * output_raw;
    pid->prev_output=*output;
    pid->derivative=ALPHA * pid->prev_derivative + (1 - ALPHA) * pid->derivative;
}


void Control_PV(PV_ControllerStates* ctrl, float setpoint, float measurement, float* output) {
//  float pos_filtered = 0.3333f * (measurement + ctrl->prev_pos) + 0.3333f * ctrl->prev_pos_filtered;
//    float vel_filtered=0.3333f*(ctrl->prev_vel_filtered)+66.67f*PV_KV*pos_filtered-66.67f*ctrl->prev_vel_input;
//    ctrl->prev_vel_input=PV_KV*pos_filtered;
//    ctrl->prev_vel_filtered=vel_filtered;
//    ctrl->prev_pos_filtered=pos_filtered;
//    ctrl->prev_pos=measurement;
//    float error = PV_KP*(setpoint - pos_filtered);
//    *output = error - vel_filtered;
//    ctrl->prev_output = *output;
    float error = PV_KP*(setpoint - measurement);
    *output = error - PV_KV*(ctrl->current_V);
    *output = ALPHA * ctrl->prev_output + (1.0f - ALPHA) * (*output);
    ctrl->prev_output = *output;

    /*
    char debug_buf[64];
    snprintf(debug_buf, sizeof(debug_buf), "PV_Output=%.2f\r\n", *output);
    UART_sendString(debug_buf);
    */
}
```

```c
LeadFilter myLead_X = {
    .b0 = 46.2f,
    .b1 = -41.8f,
    .a1 = 0.6f,
    .x_prev = 0,
    .y_prev = 0
};
LeadFilter myLead_Y = {
    .b0 = 46.2f,
    .b1 = -41.8f,
    .a1 = 0.6f,
    .x_prev = 0,
    .y_prev = 0
};
void Control_Loop(){
    ControlMode controlMode=handler->controlMode;
    UpdateTime();
    float servo_X_output=0;
    float servo_Y_output=0;
    Control_update_parameters();
    switch(controlMode){
        case MODE_PID:
        {
            Control_PID(&PID_X, desired_X, current_X,&servo_X_output);
            Control_PID(&PID_Y, desired_Y, current_Y,&servo_Y_output);

            break;
        }
        case MODE_PV:
        {
            Control_PID2(&PID_X, desired_X, current_X,&servo_X_output);
            Control_PID2(&PID_Y, desired_Y, current_Y,&servo_Y_output);

            break;
        }
        case LQR:
        {
            const float dt = handler->dt;

            Control_LQR(current_state, desired_state, u);

            Control_LQR_UpdateState(current_state, u, dt);

            servo_X_output = u[0];
            servo_Y_output = u[1];
            break;
        }
        case MODE_CUSTOM2:
        {

            LeadCompensator_Update(&myLead_X,desired_X, current_X,&servo_X_output);
            LeadCompensator_Update(&myLead_Y,desired_Y, current_Y,&servo_Y_output);
            break;
        }
        default:
             break;
    }
    Servo_X_SetAngle(servo_X_output);
    Servo_Y_SetAngle(servo_Y_output);
    snprintf(debug_buf2, sizeof(debug_buf2),
        "{\"Servo_X\":%.2f,\"Servo_Y\":%.2f,\"desired_X\":%.2f,\"desired_Y\":%.2f,"
        "\"kp_x\":%.2f,\"kd_x\":%.2f,\"ki_x\":%.2f,"
        "\"kp_y\":%.2f,\"kd_y\":%.2f,\"ki_y\":%.2f}",
        servo_X_output, servo_X_output,
        desired_X, desired_Y,
        PID_X.prop, PID_X.derivative, PID_X.integral,
        PID_Y.prop, PID_Y.derivative, PID_Y.integral);

    UART_sendString(debug_buf2);
    UART_sendString("\r\n");
    memset(debug_buf2, 0, sizeof(debug_buf2));
```

```
    HAL_Delay(17);}
void LeadCompensator_Update(LeadFilter* comp, float setpoint, float measurement, float* output) {
    float input=setpoint - measurement;
    *output = comp->b0 * input + comp->b1 * comp->x_prev - comp->a1 * comp->y_prev;
    *output = ALPHA * comp->y_prev + (1.0f - ALPHA) * (*output);
    comp->x_prev = input;
    comp->y_prev = *output;
}
float prev_setpoint_x=0.0f;
float prev_setpoint_y=0.0f;

void Control_update_parameters(){
    if(desired_X!=prev_setpoint_x || desired_Y!=prev_setpoint_y){
        newsetpoint=true;
    }
    else{
        newsetpoint=false;
    }
    prev_setpoint_x=desired_X;
    prev_setpoint_y=desired_Y;
    trajectoryMode = handler->trajectoryMode;
    if (trajectoryMode == MODE_POINT || trajectoryMode == MODE_PATH) {
        desired_X = handler->desiredX;
        desired_Y = handler->desiredY;
        PID_X.desired_V = 0;
        PID_Y.desired_V = 0;
        ctrl_x.desired_V = 0;
        ctrl_y.desired_V = 0;
    }

    tolerance = handler->tolerance;
    current_X = handler->currentX;
    current_Y = handler->currentY;

    if(handler->controlMode==MODE_PID){
        PID_X.kp = KP;
        PID_X.ki = KI;
        PID_X.kd = KD;
        PID_Y.kp = KP;
        PID_Y.ki = KI;
        PID_Y.kd = KD;
        PV_KP = (handler->PV_KP < 0.0f) ? PV_KP : handler->PV_KP;
        PV_KV = (handler->PV_KV < 0.0f) ? PV_KV : handler->PV_KV;
        KP = (handler->PID_KP < 0.0f) ? KP : handler->PID_KP;
        KI = (handler->PID_KI < 0.0f) ? KI : handler->PID_KI;
        KD = (handler->PID_KD < 0.0f) ? KD : handler->PID_KD;
        PID_X.current_V = handler->current_velocity_X;
        PID_Y.current_V = handler->current_velocity_Y;
    }
    if(handler->controlMode==MODE_PV){
        PV_KP = (handler->PV_KP < 0.0f) ? PV_KP : handler->PV_KP;
        PV_KV = (handler->PV_KV < 0.0f) ? PV_KV : handler->PV_KV;
        ctrl_x.current_V = handler->current_velocity_X;
        ctrl_y.current_V = handler->current_velocity_Y;
        ctrl_x.desired_V = handler->desired_velocity_X;
        ctrl_y.desired_V = handler->desired_velocity_Y;
    }

    if(handler->controlMode==LQR){
        current_state[0] = current_X;
        current_state[1] = handler->current_velocity_X;
        current_state[2] = current_Y;
        current_state[3] = handler->current_velocity_Y;

        desired_state[0] = desired_X;
        desired_state[1] = 0;
        desired_state[2] = desired_Y;
        desired_state[3] = 0;
    }
    ALPHA = handler->ALPHA;// Flash_Write_Control_Parameters();
}
```

```c
static uint32_t last_update_time = 0;
static float angle_rad = 0.0f;

void generate_circle_points() {
    uint32_t current_time = HAL_GetTick();
    uint32_t elapsed_time = current_time - last_update_time;

    if (elapsed_time >= (10 * (1.0f - handler->trajectroy_speed))) {
        last_update_time = current_time;

        float radius = handler->trajectroy_scale * 12;
        float angular_speed = handler->trajectroy_base_speed * handler->trajectroy_speed;

        angle_rad += angular_speed * (elapsed_time / 1000.0f);
        if (angle_rad > 2 * 3.14f) {
            angle_rad -= 2 * 3.14f;
        }

        desired_X = radius * cosf(angle_rad);
        desired_Y = radius * sinf(angle_rad);
        Control_Loop();
    }
}

void generate_infinity_points() {
    uint32_t current_time = HAL_GetTick();
    uint32_t elapsed_time = current_time - last_update_time;

    if (elapsed_time >= (10 * (1.0f - handler->trajectroy_speed))) {
        last_update_time = current_time;

        float radius = handler->trajectroy_scale * 12;
        float angular_speed = handler->trajectroy_base_speed * handler->trajectroy_speed;

        angle_rad += angular_speed * (elapsed_time / 1000.0f);
        if (angle_rad > 2 * 3.14f) {
            angle_rad -= 2 * 3.14f;
        }

        desired_X = radius * sinf(angle_rad);
        desired_Y = radius * sinf(angle_rad) * cosf(angle_rad);
        Control_Loop();
    }
}

void Control_Reset(){
    float kp = PID_X.kp;
    float ki = PID_X.ki;
    float kd = PID_X.kd;
    PID_X = (PIDController){
        .kp = kp, .ki = ki, .kd = kd,
        .integral = 0.0f,
        .tolerance = 0.0f,
        .desired_V = 0.0f,
        .current_V = 0.0f,
        .prev_output = 0.0f
    };

    PID_Y = (PIDController){
            .kp = kp, .ki = ki, .kd = kd,
            .integral = 0.0f,
            .tolerance = 0.0f,
            .desired_V = 0.0f,
            .current_V = 0.0f,
            .prev_output = 0.0f
        };
    Control_PV_Init();
    LQR_Init(&ctrl);
}
```


GND_S
6V
PA1
GND
VCC
SIG
SERVO
MG996R
Y-AXIS
servo_MG996R
GND_S
6V
PA0
GND
VCC
SIG
SERVO
MG996R
X-AXIS
servo_MG996R
CAM_FOR_BALL/LASER_DETECTION
USB_CAMERA
LAPTOP_Port
USB
J2
J1
FTDI USB-TTL Cable
GND
CTS
VCC
TXD
RXD
RTS
PA3
PA2
U2
STM32F103C8T6
PA0-WKUP
PA1
PA2 TX
PA3 RX
PA4
PA5
PA6
PA7
PA8
PA9 RX
PA10 TX
PA11
PA12
PA13 SWDIO
PA14 SWCLK
PA15
PC13-TAMPER-RTC
PC14-OSC32_IN
PC15-OSC32_OUT
PD0-OSC_IN
PD1-OSC_OUT
RX PB0
TX PB1
PB2 BOOT1
PB3
PB4
PB5
PB6
PB7
PB8
PB9
TX PB10
RX PB11
PB12
PB13
PB14
PB15
BOOT0
NRST
BAT
VCC(3.3)
VCC(3.3)
VCC5V
GND1
GND2
GND3
VSSA
VDDA
GND
VCC
H1
BUCK_1
VCC_IN
GND
GND_SERVO
SERVO_6V
GND_S
6V
TITLE:
Sheet_1
REV: 1.0
EasyEDA
Company: Your Company
Sheet: 1/1
Date: 2025-05-19
Drawn By: mar022