Self-Balancing Ball and Beam System using PID Control

Final Project Report

Team: Vincent Brunn and Jiaxing Wang

Date: April 29, 2025

1. Objective

The primary goal of this project is to design, construct, and implement a closed-loop feedback system capable of automatically balancing a small wheeled cart at a specific, user-defined position along a tilting beam. The system must react to disturbances and actively correct the cart's position using sensor data and PID control logic.

2. Project Description

This project tackles the classic "ball and beam" control challenge. We constructed a system comprising a beam, pivoted at its center, upon which a small, four-wheeled cart can roll freely along one dimension. The core task is to automatically adjust the angle of this beam to precisely control the cart's position.

An Ultrasonic Sensor (HC-SR04) is mounted at one end of the beam to continuously measure the distance to the cart. This real-time position data serves as the input (Process Variable) for our control system. An Arduino Uno microcontroller processes this sensor data. It compares the measured distance to a target distance (Setpoint), which the user can dynamically adjust using a Potentiometer.

The difference between the measured and target distance constitutes the error signal. This error is fed into a Proportional-Integral-Derivative (PID) control algorithm implemented on the Arduino. The PID algorithm calculates the necessary corrective action – specifically, the desired angle or, more directly, the target position for the motor controlling the beam's tilt. The Arduino translates the PID output into step and direction signals for an A4988 Stepper Motor Driver. The driver amplifies these signals to power a NEMA 17 Stepper Motor. The motor is mechanically coupled to the beam (via a crank/shaft linear system), allowing it to make precise adjustments to the beam's tilt angle.

This continuous cycle of measuring the cart's position, calculating the error, determining the control action via PID, and adjusting the beam angle via the motor forms a closed-loop control system. The system actively works to minimize the position error and maintain the cart at the desired location, compensating for minor disturbances or changes in the setpoint. The physical structure, including the beam, pivot, cart, and component housings, was custom-designed and 3D-printed to integrate all parts effectively.

3. Components Required

The following components were utilized in the construction and operation of the system:

- **Microcontroller:** Arduino Uno R3 (Brain of the system)
- **Sensor:** HC-SR04 Ultrasonic Distance Sensor (Measures cart position)
- **Actuator:** NEMA 17 Stepper Motor (Provides force to tilt the beam)
- **Motor Driver:** A4988 Stepper Motor Driver Module (Interfaces Arduino with the stepper motor)
- **Driver Interface:** Arduino CNC Shield V3 (Provides convenient connections for A4988 and motor)
- **Setpoint Input:** 10kΩ Linear Potentiometer (Allows user to set the target position)
- **Power Supply:** 12V DC Power Adapter (Supplies power primarily for the stepper motor via the CNC shield) and USB connection for Arduino.
- **Structure & Mechanics:**
    - 3D-Printed Beam, Pivot Mount, Cart, Motor Mount, Sensor Mount, Base/Housing
    - Linear Guides (For the track, 50cm Wooden Dowels)
    - Bearings (For pivot point and cart wheels)
    - M-5 Nuts, Screws, and Washers (For assembly)
- **Wiring:** Jumper Wires

4. Component Basics Explained

- **Arduino Uno:** The microcontroller serves as the central processing unit. It reads data from the ultrasonic sensor and potentiometer, executes the PID control logic based on the programmed code, and sends precise step and direction commands to the A4988 motor driver to adjust the beam's angle.
- **HC-SR04 Ultrasonic Sensor:** This sensor measures the distance to the cart using sound waves. It emits a short ultrasonic pulse (via the TRIG_PIN) and measures the time it takes for the echo to bounce off the cart and return (via the ECHO_PIN). The Arduino calculates the distance based on this time-of-flight measurement ($\text{Distance}=(\text{Time}\times\text{SpeedOfSound})/2$).
- **NEMA 17 Stepper Motor:** An electric motor designed for precise rotational control. It moves in discrete increments or "steps" in response to electrical pulses. This allows for accurate positioning of the beam angle without needing a separate position sensor on the motor itself (though it operates within the larger closed loop of the cart position).
- **A4988 Motor Driver:** A specialized integrated circuit module that simplifies stepper motor control. It receives simple logic-level signals (Step, Direction, Enable) from the Arduino and translates them into the higher-current, precisely timed electrical sequences required to energize the motor windings and cause rotation. It also handles microstepping, allowing for smoother motion than full steps.

- **Arduino CNC Shield:** A convenient add-on board for the Arduino Uno that provides dedicated sockets for A4988 drivers, screw terminals for motor connections, and standard pins for other inputs/outputs, simplifying the wiring process significantly compared to breadboarding.
- **Potentiometer:** A three-terminal variable resistor. By connecting the outer terminals to power and ground, and the middle terminal (wiper) to an analog input pin (POT_PIN) on the Arduino, turning the knob changes the voltage read by the Arduino. This analog reading (0-1023) is then mapped to the desired target distance (targetDist) for the cart.

5. Code Implementation

The core logic controlling the system is implemented in the Arduino code presented in the accompanying code. This code integrates sensor reading, filtering, PID computation, and motor control functions.

(See the code artifact for the full source code.)

6. Code Breakdown

1. **Includes:** Necessary libraries are included: HC_SR04.h for the ultrasonic sensor, AccelStepper.h for smooth stepper motor control, and PID_v1.h for the PID algorithm.
2. **Pin Definitions & Constants:** #define directives assign meaningful names to the Arduino pins connected to the sensor and motor driver (via the CNC Shield). Constants like NUM_SAMPLES (for the moving average filter) and UPDATE_INTERVAL (sensor ping frequency) configure system behavior.
3. **Global Variables:** Key variables are declared: sensor object, samples array and averageDist for sensor filtering, targetDist for the setpoint, input and output for the PID controller, the pid object itself, and the stepper object for motor control.
4. **setup() Function:** This runs once at startup. It initializes serial communication for debugging, configures the ultrasonic sensor for asynchronous operation (using interrupts via beginAsync), sets up the stepper motor driver pins (Enable, Step, Direction - handled by AccelStepper library) and defines motor performance parameters (setMaxSpeed, setAcceleration). Crucially, it initializes the PID controller (SetMode(AUTOMATIC), SetSampleTime, SetOutputLimits, SetTunings).
5. **loop() Function:** This runs repeatedly.
   - **Sensor Reading & Filtering:** Periodically checks if it's time for a new sensor reading (UPDATE_INTERVAL). When ready and the previous async read is done (sensor.isFinished()), it gets the distance (getDist_cm), performs basic outlier rejection (ignoring readings outside expected range, e.g., < 2 or > 52), updates a moving average filter (averageDist) to smooth the readings, and triggers the next sensor ping (startAsync).

    - **Setpoint Update:** Reads the potentiometer (analogRead(POT_PIN)) and maps its 0-1023 value to the desired operational range of the cart (e.g., 5cm to 30cm) to update targetDist.
    - **Debugging Output:** Prints the current averageDist and targetDist to the Serial Monitor.
    - **PID Calculation:** Assigns the filtered sensor reading (averageDist) to the PID input. Calls pid.Compute(), which calculates the error (targetDist - input) and determines the required control output based on PID logic and tuning parameters.
    - **Motor Control:** Uses stepper.moveTo(output) to set the target absolute position for the stepper motor based on the PID output. The stepper.run() function is called frequently to execute necessary steps towards the target position, managing speed and acceleration smoothly.

7. PID Control Explained

Concept:
PID (Proportional-Integral-Derivative) control is a feedback control loop mechanism used widely in industrial control systems and automation. It aims to minimize the error between a measured process variable (PV) and a desired setpoint (SP) by calculating and applying a corrective action. The correction is based on three terms:

- **Proportional (P) Term (**$Kp \cdot e(t)$**):** Responds to the *present* error. The larger the current difference between SP and PV, the larger the corrective action. Kp is the proportional gain.
- **Integral (I) Term (**$Ki\int e(t)dt$**):** Responds to the *past* error. It accumulates the error over time, driving out steady-state errors that the P term alone might not overcome (e.g., due to constant friction). Ki is the integral gain.
- **Derivative (D) Term (**$Kddtde(t)$**):** Responds to the *future* error based on its current rate of change. It acts as a damper, slowing down the response as the PV approaches the SP, reducing overshoot and oscillations. Kd is the derivative gain.

The controller output is the sum: $Output=Kpe(t)+Ki\int e(t)dt+Kddtde(t)$, where $e(t)=SP-PV$.

Application and Calibration in this Project:

- **PV:** Cart position (averageDist).
- **SP:** Target position (targetDist).
- **Error** $e(t)$**:** targetDist - averageDist.
- **Output:** Motor target position (output variable, in steps). The PID output directly commands the desired motor position needed to tilt the beam appropriately.
- **Calibration (Tuning):** This involves finding optimal values for Kp, Ki, and Kd (set via pid.SetTunings(Kp, Ki, Kd)). This is critical for performance. The process typically involves:
    1. Setting $Ki=0,Kd=0$ and increasing Kp until stable oscillations occur, then reducing Kp.
    2. Increasing Ki to eliminate steady-state error, watching for overshoot/oscillation.

3. Increasing Kd to dampen oscillations and improve settling time, being careful not to amplify sensor noise.
   This is iterative. The values (70, 60, 50) used in the code reflect one point in this tuning process. The SetOutputLimits(-10000, 10000) prevents excessive motor commands and integral windup. The SetSampleTime(30) determines how often the PID recalculates, balancing responsiveness with processing load.

8. Outcome

The project successfully resulted in a functional self-balancing beam system. The cart's position is actively monitored by the ultrasonic sensor, and the Arduino, executing the PID algorithm, commands the stepper motor to tilt the beam, effectively maintaining the cart near the user-defined setpoint adjusted via the potentiometer. The system demonstrates the principles of closed-loop control and PID regulation. It generally holds the position well but exhibits some minor oscillations or slow settling depending on the PID tuning and the specific setpoint. A video demo can be found here:
https://drive.google.com/file/d/1FnhjYiGbXrUGmqKmbNPuyDkxjPleGBRv/view?usp=sharing

Strengths:

- Successful implementation of a real-time control system.
- Practical demonstration of PID control theory.
- Integration of sensor, microcontroller, and actuator.
- User-adjustable setpoint via potentiometer.

Potential Shortfalls & Areas for Improvement:

- **Sensor Limitations:** The HC-SR04's accuracy, noise level, update rate, and potential for outlier readings limit the ultimate precision and responsiveness, particularly affecting the D term.
- **Mechanical Factors:** Friction in the pivot and cart/track interface, potential backlash in the motor linkage, and beam flexibility introduce non-linearities and disturbances that challenge the linear PID controller.
- **PID Tuning:** The current PID parameters (70, 60, 50) provide functionality but may not be optimal across the entire operating range or for rapid disturbance rejection. Further fine-tuning could enhance performance (reduce overshoot, quicken settling time).
- **Dynamic Response:** Limited by motor speed/acceleration and control loop timing.

Difficulties Encountered & Lessons Learned:

Several challenges were overcome during development. Initial mechanical assembly required adjustments (filing 3D prints) to ensure proper fit and minimize binding, emphasizing the need for careful design tolerances. Unaccounted friction in the structure and cart movement initially hindered smooth operation and complicated control efforts, requiring mechanical refinement and robust PID tuning. Interfacing the stepper motor using the A4988 driver and CNC shield proved time-consuming, demanding careful attention to wiring, pin configuration within the Arduino code (matching shield connections), and understanding the AccelStepper library's functions for smooth motion control.

The most iterative and conceptually challenging part was tuning the PID controller. Finding the right balance between Kp, Ki, and Kd to achieve stability, responsiveness, and minimal steady-state error required significant trial-and-error. Observing the system's behavior (using Serial Plotter for averageDist and targetDist was invaluable) while adjusting the gains provided practical insight into how each PID term affects the system dynamics. This highlighted that PID tuning is highly specific to the physical system's characteristics (inertia, friction, sensor noise, actuator limits).

Code:

```
/*

  Ultrasonic-only closed-loop stepper (A4988 + NEMA17)

  STEP 2, DIR 5, EN 8

  HC-SR04:  TRIG 4,  ECHO 3  (INT1)

*/



#include <HC_SR04.h>

#include <AccelStepper.h>

#include <PID_v1.h>
```

```
/* ----------  HC-SR04  --------------------------------------- */

#define TRIG_PIN 4

#define ECHO_PIN 3                          // INT1  (pin 3)



HC_SR04<ECHO_PIN> sensor(TRIG_PIN);         // echo pin is template param



const uint8_t NUM_SAMPLES = 14;

int16_t  samples[NUM_SAMPLES] = {0};

uint8_t  sampleIndex = 0;

double   averageDist = 0;                   // filtered cm

const uint16_t UPDATE_INTERVAL = 25;        // ms between pings

unsigned long prevPingMs = 0;



/* ----------  PID  ------------------------------------------- */

double targetDist = 10

;                     // cm, trimmed by pot

double input, output;                       // cm, steps

PID pid(&input, &output, &targetDist, 1, 0, 0, DIRECT);



/* ----------  A4988 Stepper  -------------------------------- */

#define EN_PIN   8

#define STEP_PIN 2
```

```
#define DIR_PIN  5

AccelStepper stepper(AccelStepper::DRIVER, STEP_PIN, DIR_PIN);



/* ----------  Potentiometer  --------------------------------- */

const uint8_t POT_PIN = A3;



/* ============================================================ */

void setup() {

 Serial.begin(9600);

 while (!Serial) {}



 /*  HC-SR04  */

 sensor.beginAsync();                     // attach interrupt

 sensor.startAsync(60000);                // first ping (60 ms timeout)







 /*  Stepper driver  */

 pinMode(EN_PIN, OUTPUT);

 digitalWrite(EN_PIN, LOW);               // enable A4988

 stepper.setMaxSpeed(400000);               // steps s⁻¹  (safe but quick) 200000

 stepper.setAcceleration(40000);           // steps s⁻² 20000
 /*  PID  */
 pid.SetMode(AUTOMATIC);
 pid.SetSampleTime(30);                   // ms
 pid.SetOutputLimits(-10000, 10000);          // ±800 steps window
 pid.SetTunings(70, 60, 50);  // set the PID tuning parameters (P, I, D) original:
70,14,165, WORKING: 70, 28, 10
}
// 1: 50,28,10. 2: 50,28,50 3: 50,56,50, 4: 70, 56, 50 5: 70,60,50
/* ============================================================ */
void loop() {


 /* ----  Ultrasonic read every UPDATE_INTERVAL ms  ---------- */
 unsigned long now = millis();
 if (now - prevPingMs >= UPDATE_INTERVAL) {
   prevPingMs = now;


   if (sensor.isFinished()) {                     // async ping done?
     long cm = sensor.getDist_cm();               // read result


     if (cm > 52) cm = 16;                        // brutal outlier kill


     sensor.startAsync(60000);                    // launch next ping
```

```
      /* moving-average low-pass */
      samples[sampleIndex] = cm;
      sampleIndex = (sampleIndex + 1) % NUM_SAMPLES;
      long sum = 0;
      for (uint8_t i = 0; i < NUM_SAMPLES; ++i) sum += samples[i];
      averageDist = sum / (double)NUM_SAMPLES;

      /* knob → new set-point */
      targetDist = map(analogRead(POT_PIN), 0, 1023, 5, 30);

      Serial.print(averageDist);
      Serial.print(',');
      Serial.println(targetDist);
    }
  }
  /* ----  PID + stepper  ------------------------------------ */
  input = averageDist;
  pid.Compute();                          // updates 'output'
  stepper.moveTo(output);                 // closed-loop actuation
  stepper.run();
}
```