Allen, John, Lulu

Due: October 28th, 2024

PID Ball Balance

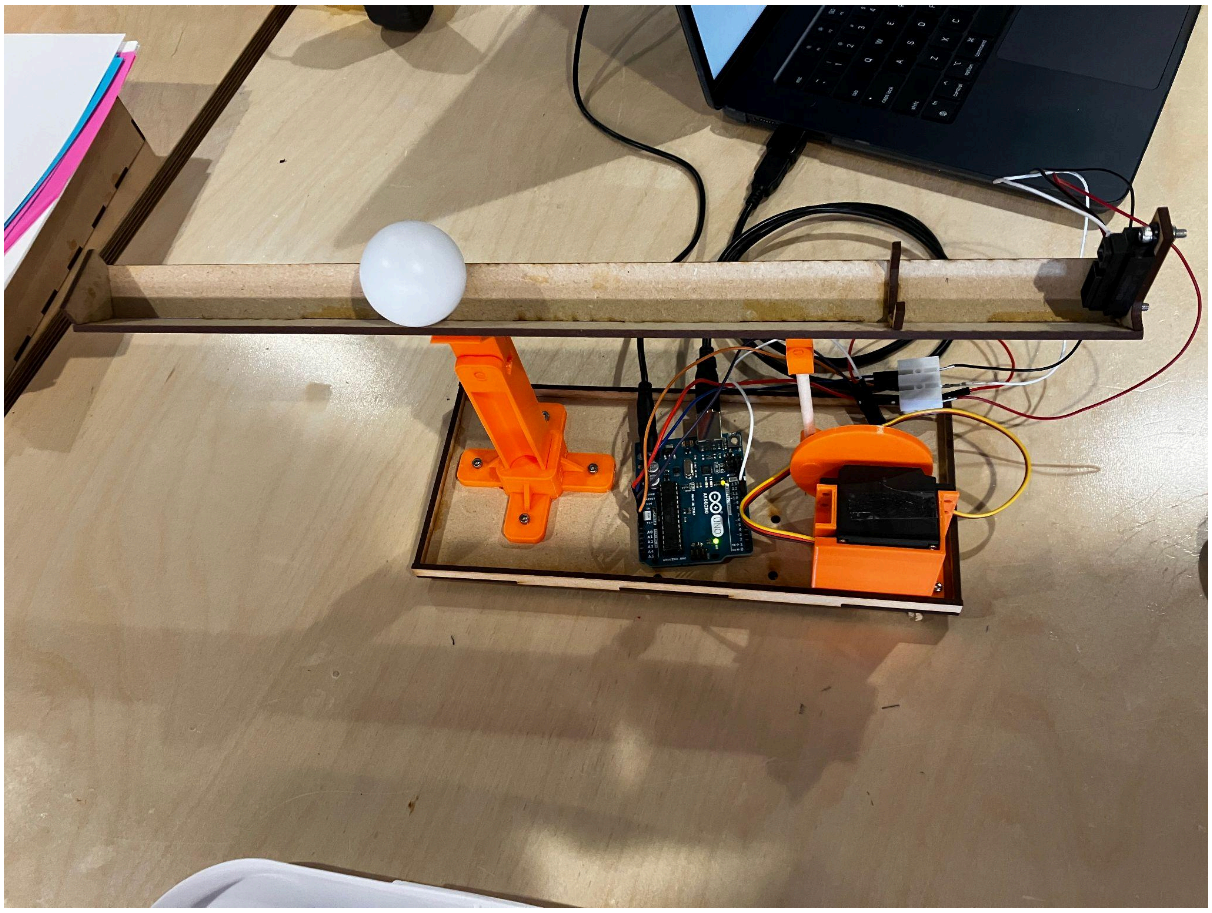
Introduction:

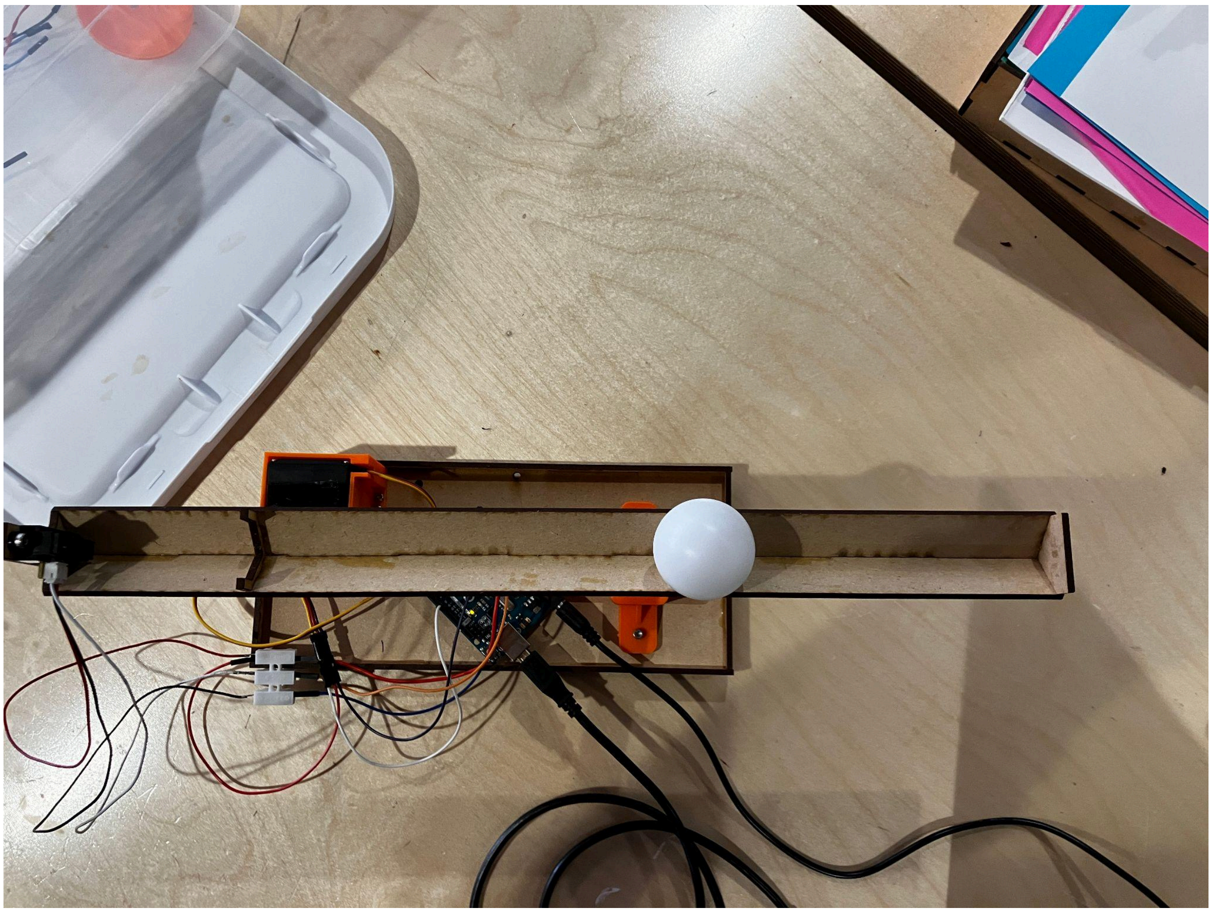
This project uses an Infrared Proximity Distance Sensor and a servo motor to implement a PID control system to balance a ping pong ball on a beam. The goal is to maintain the ball's position at the center of the beam by adjusting the servo angle based on real-time distance measurements from the sensor.

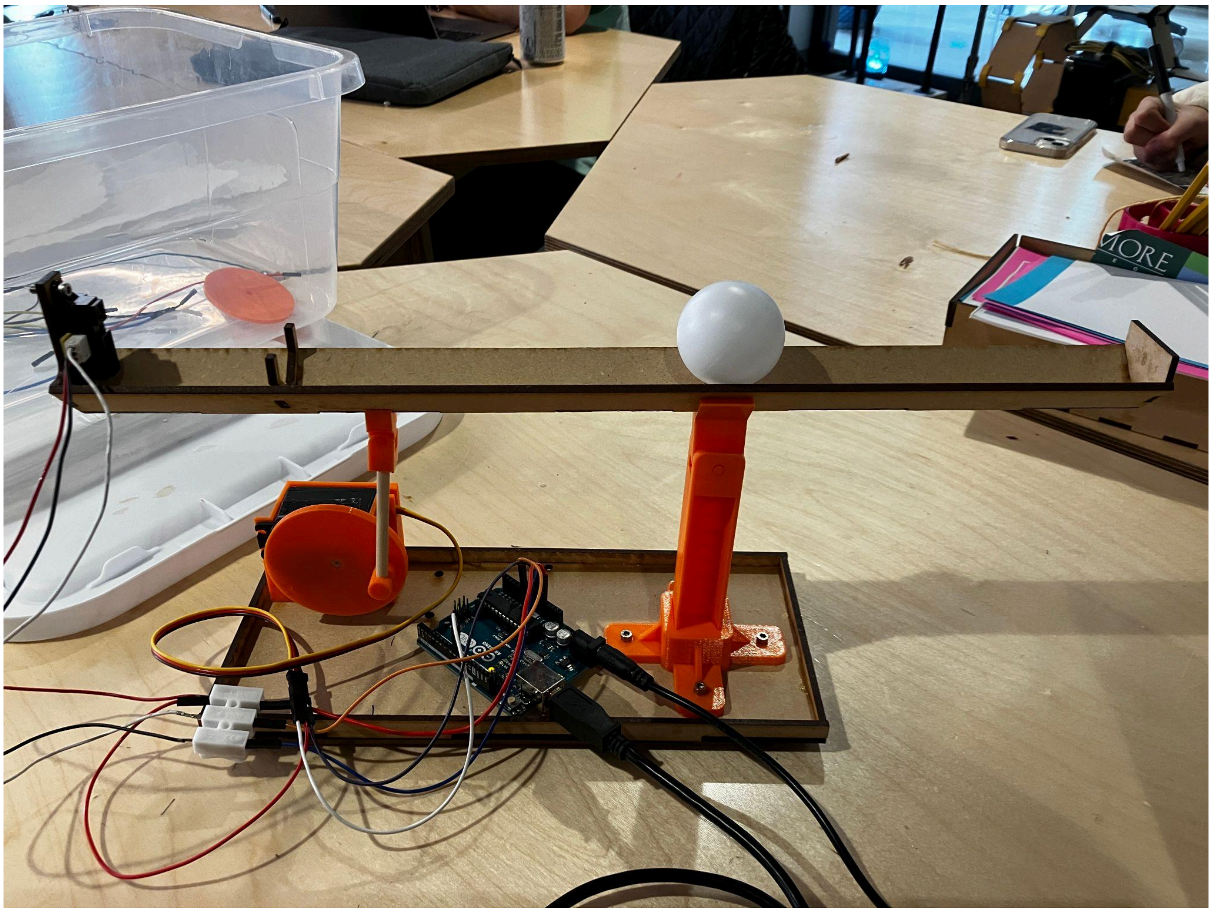
Parts List:

- Infrared Proximity Distance Sensor (Model GP2Y0A21YK0F)
- Servo motor
- 1 Arduino board
- Wires
- Ping pong ball
- M3 screws (for assembling parts)
- screws of various lengths and nuts
- 3D-printed construction parts
- External Power
- Wood glue
- Super glue

Photos of robot:







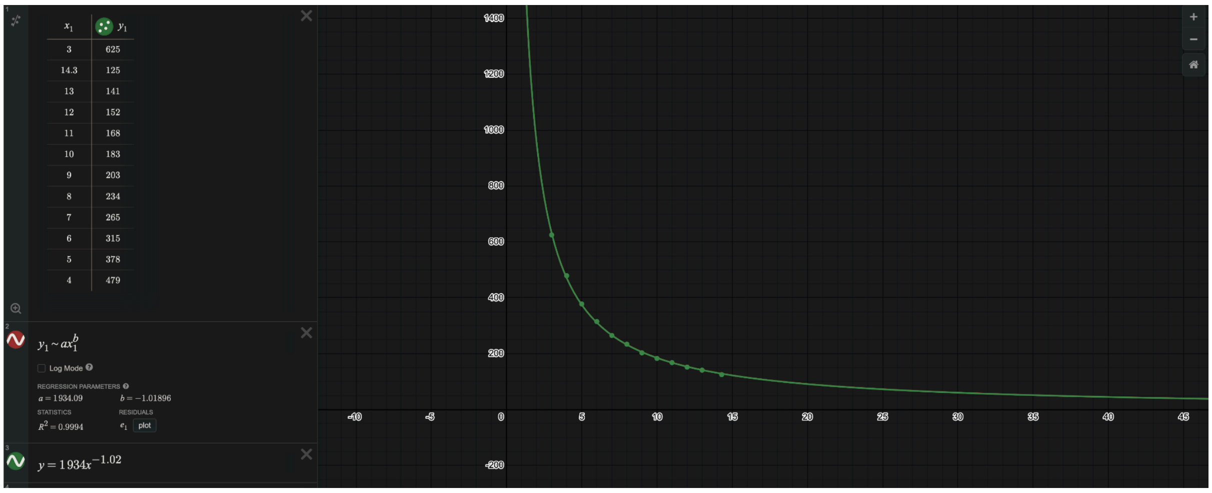
Design Procedures & Decisions:

1. Construction: we started by assembling the structure according to the handout image, but encountered two unexpected obstacles. First, we accidentally reversed the assembly of the wooden base, which prevented the edge blocks from effectively stabilizing the base by offsetting screw penetration. To avoid reprinting the entire base, we opted to glue discarded wood blocks underneath to even out the screw height. Second, the 3D-printed servo-beam joint could not move freely. We quickly identified the hardware issue, sought assistance from Professor Halstead, and promptly reprinted the part to resolve the problem. In the process of trying to tune our PID feedback, we encountered connection and power issues with our sensor and Arduino. We were able to solve this by replacing both the connecting wires to our IPDS sensor and our Arduino.
2. Calibration: to calibrate the infrared proximity distance sensor (Model GP2Y0A21YK0F), we began by writing an Arduino sketch to output raw sensor

readings (also, this step tested the functionality of the sensor). Starting with the ball at the bracket's position (which is 3 inches from the sensor as we measured), we used a ruler alongside the beam to measure the distance and gradually move the ball in approximately 1 inch increments away from the sensor. At each point, we recorded the sensor's output, creating a detailed dataset to map voltage readings to distances(see Table 1). Next, using Desmos, we plotted the sensor output and fit the data to a power law function $y = ax^b$, where a = 1934.09 and b = −1.01896. This function provided an excellent fit, with a R^2 value of 0.9994, indicating high accuracy. Finally, we rearranged the formula and simplified our values to express distance as a function of voltage: $x = \frac{1934^{0.98}}{y^{0.98}}$, allowing us to accurately calculate the distance from any voltage reading (see Figure 1). We found our balance point to be roughly 8.54 inches from the sensor.

	14.3 cm	13 cm	12 cm	11 cm	10 cm	9cm	8cm	7cm	6cm	5cm	4cm	3cm
V(analog)	125	141	152	168	183	203	234	265	315	378	479	625

Table 1: Calibration Data: Distance vs. Voltage Output of the Sensor



x_1	y_1
3	625
14.3	125
13	141
12	152
11	168
10	183
9	203
8	234
7	265
6	315
5	378
4	479

$y_1 \sim a x_1^b$

Log Mode

REGRESSION PARAMETERS

$a = 1934.09$ $b = -1.01896$

STATISTICS

$R^2 = 0.9994$

RESIDUALS

e_1 plot

$y = 1934x^{-1.02}$

Figure 1: Calibration Curve: Voltage to Distance Mapping with Power Law Fit

3. Servo Control: to control the servo motor, we used the `Servo.h` library as suggested in the handout. We started by including the library and creating a servo object using `Servo myservo;`. In the `setup()` function, we attached the servo to pin 9 on the Arduino using `myservo.attach(servoPin);` and set it to the initial position of 90 degrees with `myservo.write(90);`. This allowed us to place the beam in a neutral position at the beginning. In our `adjustHeight()` function, we used `myservo.write(angle);` to dynamically adjust the servo angle based on the PID control calculations, which was important for precise control over the angle to balance the ping pong ball effectively on the beam. We utilized the `constrain()` function to ensure the angle stayed within the 0 to 180-degree range, which prevented the servo from attempting to move beyond its physical limitations.
We used the exponential filter with a weight of 0.2 to filter out the noise from our IPDS readings, which helped reduce jitters and improve our adjustment process. We also normalized our error between -1 and 1 in our `adjustHeight()` function to make it easier to work with.

4. PID Control: aiming to maintain the ball's position at the center of the beam, we implemented a PID feedback loop to control the servo angle based on the sensor's distance readings.
   - Proportional Control (P): the proportional adjustment, `up`, was calculated using the difference between the target set point and the sensor reading, scaled by a gain factor `K`. It provided immediate correction based on the error.
   - Integral Control (I): we accumulated past errors over time into `cumError`, which was scaled by `K/Ti` to yield `ui`, in order to address steady-state error. We reset `cumError` if the error direction changed to prevent excessive buildup.
   - Derivative Control (D): the derivative adjustment, `ud`, calculated the rate of change of error to dampen oscillations. It is done by finding the difference between the current and previous errors, divided by the elapsed time `dt`.

   The total control signal, `u`, combined `up`, `ui`, and `ud`, centered around 90 degrees to set the servo angle. Using `constrain()`, we limited `u` to 0-180 degrees to match the servo's range, which allows the beam to move stably.

Code:

```
#include <Servo.h> //Include the servo library

const int IPDS_Sensor = A0; //Infrared sensor attached to analog pin A0
const int servoPin = 9; // Servo attached to digital pin 9

const int setPoint = 8.54; // Approximately the distance measurement that should be read when the ball is in the center of the beam

Servo myservo; // Initialize the servo.
```

```
float prevMeasure; //Initialize previous measurement value used in exponential
filter.

//Initialize time values used in adjustHeight function
long previousTime;
long currentTime;

long previousError; // Initialize previousError value used for adjustment function

float error; // Initialize error. Used in adjustment function.
float angle; // Initialize the angle. This is what is used by the adjustment
function to set the servo angle.

float dt; //The time elapsed between loops

float u; // Total adjustment
float up; //Proportional adjustment
float ui; //Integral adjustment
float ud; //Derivative adjustment

float cumError = 0; //Initialize cumulative error to 0.



void setup() {
   Serial.begin(9600); // Set baud rate
   myservo.attach(servoPin); //Initialize servo for function calls
   myservo.write(90); //Test the servo and give it an initial value.
   previousTime = millis(); //Initialize the previousTime value for the first call.
   delay(500); //Wait before initiating balancing loop.
}


void loop() {
 float raw_measure = (pow(1934.09, 0.98))/(pow(analogRead(IPDS_Sensor),0.98));
//Get the distance measurement from the proximity distance sensor. The constant
values were found from the calibration process for our sensor.
 delay(5); //Wait a small amount of time between measurements.

 float weight = 0.2; // The weighting value used for the exponential filter
 float sensorDistance = (weight * raw_measure) + (1 - weight) * prevMeasure; //
Filter out noise from the measurements using exponential filter.
 prevMeasure = sensorDistance; // Set the previous measurement to the current one
for use in the exponential filter the next time around
```

```
 adjustHeight(sensorDistance, setPoint, 10.7, 4.8, 0.01); // Call the adjustment
function, passing in the constants found in the process of tuning

}



/* Function used for adjusting the servo output. Takes 5 parameters:
   sensorValue: The current measurement from the proximity sensor.
   set_point: The objective distance for the ball
   K: Constant value used in adjustment.
   Ti: Integral constant value.
   Td: Derivative constant value.
 Sets the servo to the appropriate value for balancing the ball.*/
float adjustHeight(float sensorValue, float set_point, float K, float Ti, float
Td){
 error = set_point - sensorValue; // Get the error as the difference between the
goal distance and the actual distance measured
 error = error / 5.0; //(roughly) Normalize error between -1 and 1

 cumError += error; //Add error to the running cumulative error.

 if((cumError > 0 && error < 0) || cumError < 0 && error > 0){
   cumError = 0; //If the ball overshoots the midpoint, reset the cumulative error
in order to limit integral buildup
 }

 currentTime = millis(); //Get the current time
 dt = (currentTime - previousTime)/1000.0; //Calculate how much time has elapsed
and convert to seconds
 previousTime = currentTime; // Set the previous time to the current time for the
next loop

 up = K * error; //Set the proportional adjustment
 ui = (K/Ti) * cumError * dt; //Set the integral adjustment
 ud = K * Td * (error-previousError)/dt; //Set the derivative adjustment
 previousError = error; //Set the previous error to the current error for the next
adjustment step

 u = up + ui + ud + 90; //Set the total adjustment to the proportional adjustment +
the integral adjustment + the derivative adjustment + 90. The 90 is so that when
the ball is stationary in the middle of the beam, the servo value is set to 90
which keeps the beam perfectly flat
 angle = constrain(u, 0, 180); //Constrain the angle adjustment between 0 and 180,
the range of values accepted by the servo
 myservo.write(angle); //Write the angle for balancing to the servo
```

```
}
```

In conclusion, our balance-bot works pretty well. It is a bit slow to adjust as the ball gets closer to being perfectly in the center, but this tradeoff allows the robot to consistently get the ball fairly close to the center point quickly. Our range of motion limitations also mean that though the integral adjustment will continue to adjust the ball, there is not enough spare angle for it to roll over obstacles. With more time we might continue tuning our adjustment constants and trying different combinations of influence. Overall we are happy with how the robot turned out.

References

Robotics_Assignments_PIDBalance.pdf (n.d.)

https://thespring.skidmore.edu/d2l/common/viewFile.d2lfile/Database/MTc1ODM2OA/Robotics_Assignments_PIDBalance.pdf?ou=55798