**ENME/ENEE 461**

**Controls Project - Self-Stabilizing Robot**

**Stephen Snyder, Naitik Gupta**

**Introduction**

**Goal**

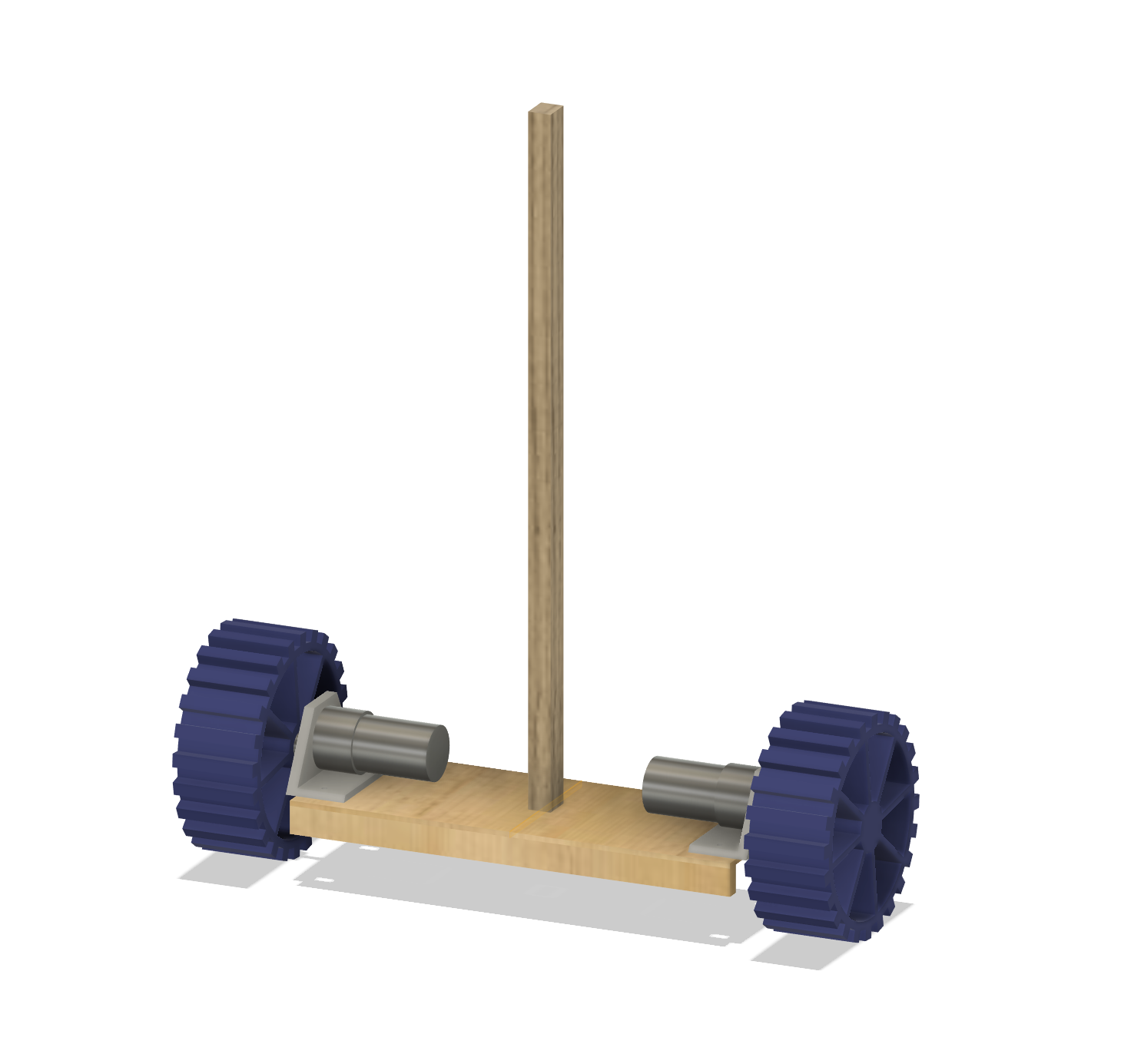
Our main goal for this project is to apply our knowledge and learnings from this semester into a real world application that presents our mastery of the course. For this, we chose to design a self-balancing robot, similar in design to a Segway. We got our inspiration from the inverted pendulum project where we had to design a PD controller that kept a metal pendulum upright using a metal arm that spins. In our case, we used two 12v motors that were fast and had enough torque to keep the robot from tipping over too much.

**Hypothesis**

Using a PD controller with an accelerometer that calculates the change in angle as input is a viable way to control two motors with varying speeds to keep a self-balancing robot upright.

**Design**

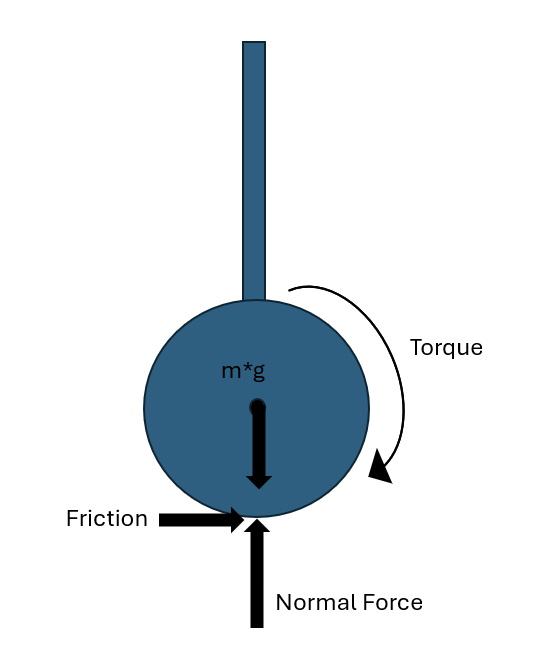
**CAD Model**



*Fig 1: CAD model of the robot*

Our robot was designed in Fusion360. It consists of two motors mounted to a baseplate, wheels on the ends, and a stick in the middle. During the fabrication process, extra support pieces were added to the underside of the base to keep it from bowing, which has been reflected in the CAD model.

**Motor Justification**



*Fig 2: Free-body diagram of the robot*

We derived the necessary torque per motor using a rough force balance of the vehicle and the motor torque, assuming a friction coefficient of ~0.3, a wheel radius of 130mm, and a weight of 1kg.

This worked out to be 0.191 N-m (1.95 kg-cm) of torque per motor in order to make our robot move.

**Parts Used**

We used two 12V DFRobot motors. They had a max speed of 251RPM and a stall torque of 18 kg-cm, which was enough for the motors to be able to spin even when the voltage applied was well under 12V.

Two motor mounting brackets designed by us were printed in PLA to connect the motors to the baseplate.

A Pololu MD03A Dual H-Bridge was use to control the power sent to the motors. An H-bridge allows us to control both the direction of the motors and the speed, which is necessary for our controller to function.

A 12V, 3000mAh rechargeable Li-ion battery was used as our power supply for the motors. This had the voltage and the current discharge rate necessary to power our motors, while also being lightweight. A kill switch was soldered into the power output wire to allow us to quickly shut down the robot.

A buck converter was used to step down the voltage from 12V to 5V to provide power to the accelerometer and H-bridge.

We intended to use an ESP32 as our microcontroller due to its 240MHz response speed which would provide enough speed to react to changing angle values. However, the ESP32 gave us a lot of problems. Specific drivers needed to be found for it to even interact with our computer(silicon labs cp210x USB to UART drivers). Then, the boot button needed to be held every time we tried to upload code. The COM port kept switching every time we tried to upload the code, as did the board drivers in arduino: we used ALKS 4D sometimes, 4d Systems drivers other times. Finally, once the code was able to be uploaded, the GPIO pins outputted negative voltage signals and could not switch H-bridge logic, so we decided not to use it.

We instead used an Arduino Uno, and even with a slower speed of 16MHz it was fast enough to effectively control the robot.

An Adafruit MPU6050 6-DOF accelerometer was used to get angle readings. It was mounted low to limit noise, as close to the axles of the motor around which the robot would primarily turn, and readings from only the X-direction were used.

Our wheels were 130mm in diameter and were printed in TPU to give it better grip. We wanted the wheels to be large to accommodate our circuits and battery without throwing the rover too off balance. 130mm enabled us to fit the battery at the bottom and the rest of the circuits over the axle which gave us just enough slight imbalance for our PID to work and self balance. We later wrapped the wheels in duct tape to give it even more grip.

A small PLA wheel hub with a set screw was used to connect the motor shaft to the wheels.

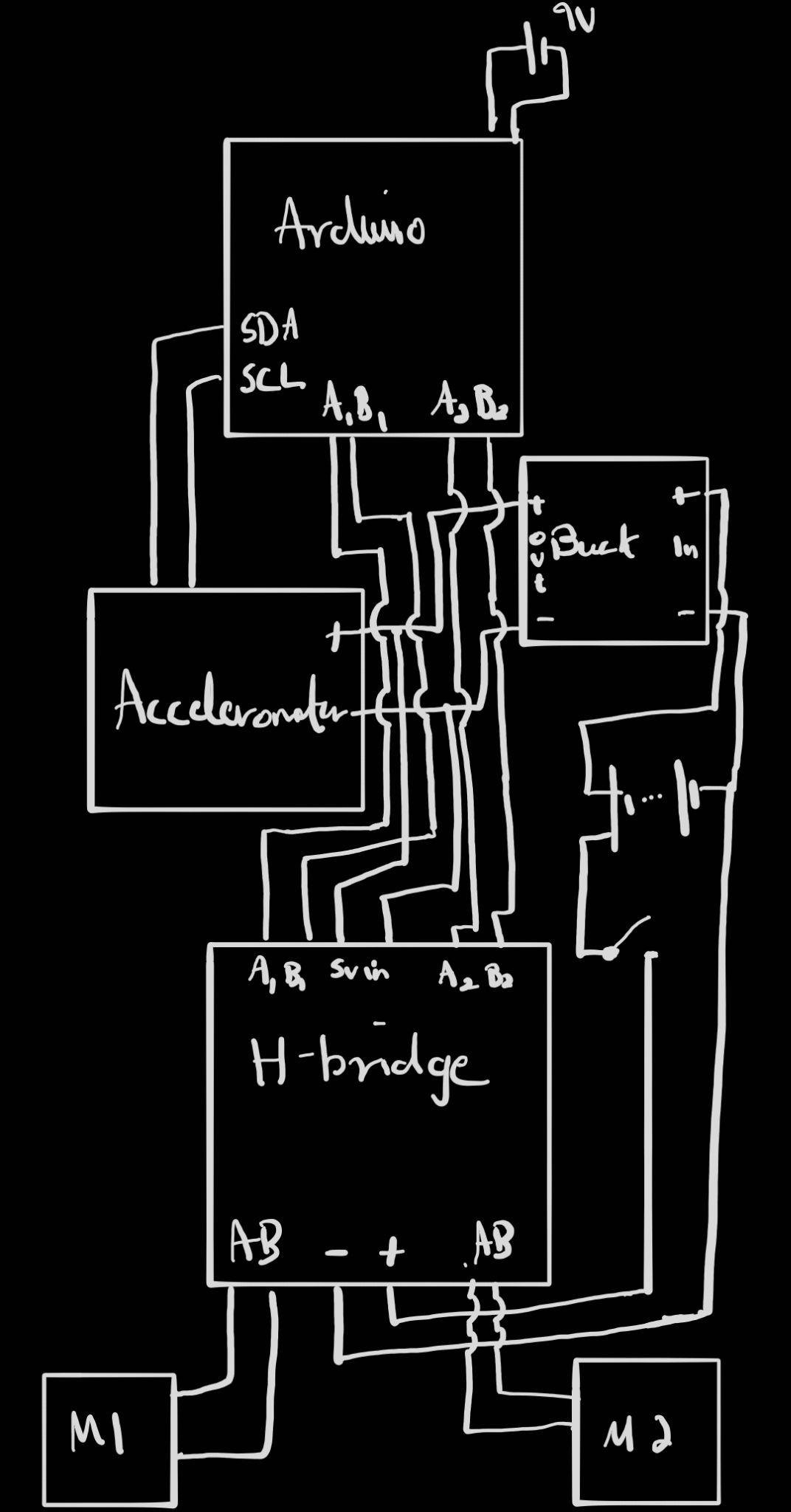
A piece of 1/8" (3.175mm) plywood was laser cut to 100x275mm as the baseplate.

A scrap piece of wood, roughly 14x18x420mm, was placed vertically in the center both to give a clear indicator of the current angle and to add enough weight above the robot to unbalance it.

Two pieces of scrap wood, roughly 8x17x275mm, were used to support the baseplate from bowing.

With the exception of the battery, which was mounted to the underside to keep it more secure, all parts were mounted on the baseplate. As much of the weight as possible was placed above the axis of rotation to cause the robot to be unstable while at rest.

**Wiring diagram**

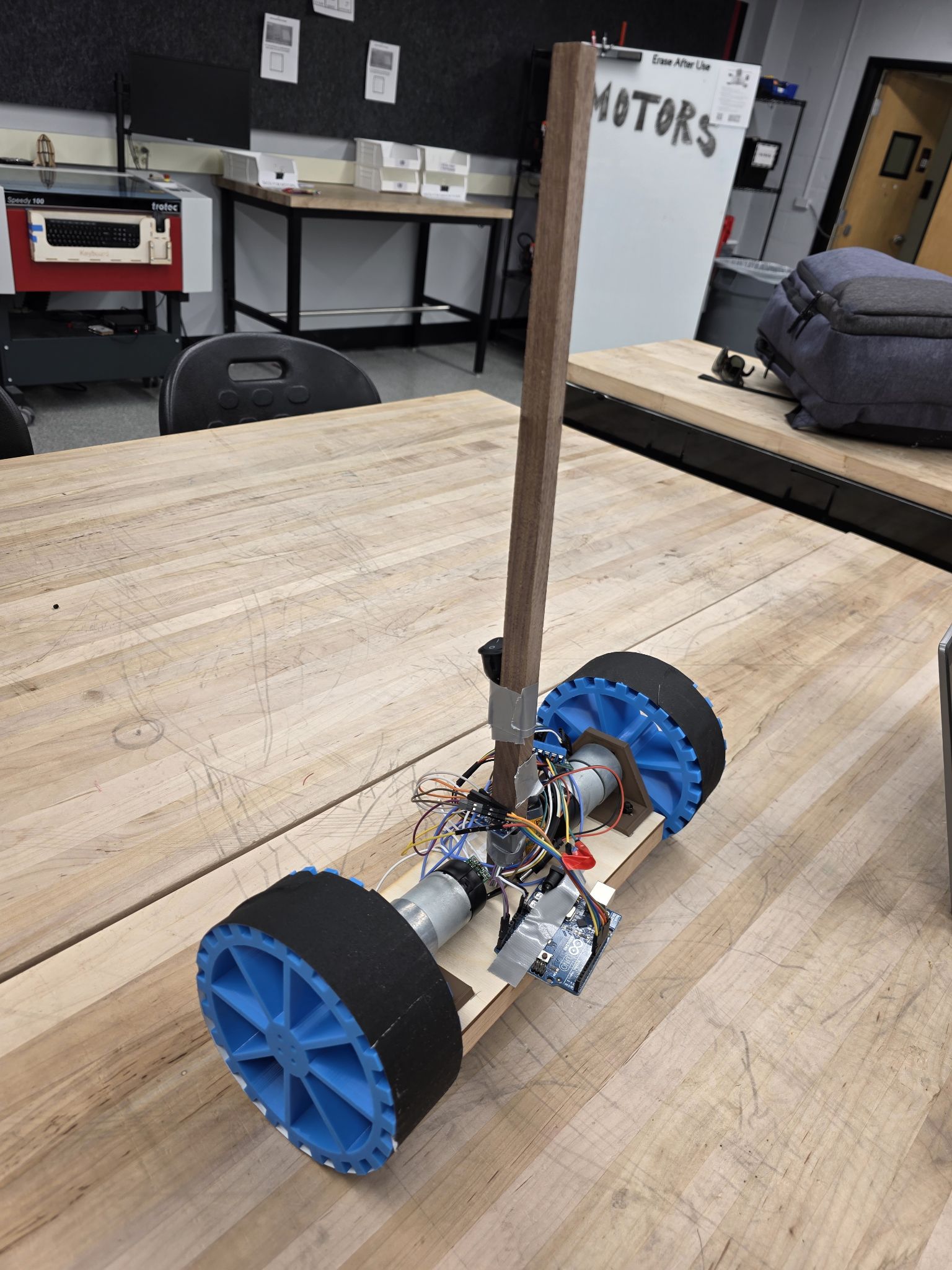
****

*Fig 3: Wiring diagram*

**Pinout**

| **Function** | **Component** | **Arduino Pin** | **Direction** | **Notes** |
| --- | --- | --- | --- | --- |
| Motor A DIR 1 | H-Bridge INA1 | 7 | Output | Sets forward/reverse |
| Motor A DIR 2 | H-Bridge INB1 | 8 | Output | Sets forward/reverse |
| Motor A PWM | H-Bridge PWM1 | 5 | PWM Output | Sets speed |
| Motor B DIR 1 | H-Bridge INA2 | 9 | Output | Sets forward/reverse |
| Motor B DIR 2 | H-Bridge INB2 | 10 | Output | Sets forward/reverse |
| Motor B PWM | H-Bridge PWM2 | 6 | PWM Output | Sets speed |
| I²C SDA | MPU6050 | A4 | I²C | Connects to MPU6050 SDA |
| I²C SCL | MPU6050 | A5 | I²C | Connects to MPU6050 SCL |
| 5V Logic Power | H-Bridge / MPU6050 | - | Power | From 5V buck converter |
| GND | All Devices | - | Ground | Common ground for all |
| VIN (Arduino) | 9V Battery | VIN | Power Input | Powers Arduino Uno |
| **Function** | **Component** | **Arduino Pin** | **Direction** | **Notes** |

**Final Design**

*****Fig 4: Fully built robot*

**Code**

**Initial Calibration Code**

We first tested the output from the accelerometer using the code below, found as an example within the accelerometer library. This outputs the angle in the X direction, the acceleration in the Y direction, and the gyroscopic force in the Z direction. The angle value was the least noisy of the three, and the X direction allowed for simple mounting to the stick, so we used that value for the remainder of the code. This code also uses the mpu.calcOffsets() function, which ensures that all values are referenced from the starting position of the accelerometer. This means we had to hold the robot in the vertical position when starting the code, in order to properly reference the zero angle point to vertical.

**#include <Wire.h>**

**#include <MPU6050\_light.h>**

**MPU6050 mpu(Wire);**

**void setup() {**

**Serial.begin(9600);**

**delay(1000);**

**Wire.begin(); // On Uno: SDA = A4, SCL = A5**

**byte status = mpu.begin();**

**Serial.print("MPU6050 status: ");**

**Serial.println(status);**

**while (status != 0) {**

**delay(1000);**

**Serial.println("MPU not connected...");**

**}**

**Serial.println("Calibrating gyro...");**

**delay(1000);**

**mpu.calcOffsets(); // gyro and accel calibration**

**Serial.println("Done!");**

**}**

**void loop() {**

**mpu.update();**

**// Angle in X-axis (pitch) for balancing**

**Serial.print("Angle X (Pitch): ");**

**Serial.print(mpu.getAngleX());**

**Serial.print("\t");**

**// Raw acceleration and gyro**

**Serial.print("Accel Z: ");**

**Serial.print(mpu.getAccZ());**

**Serial.print("\tGyro Y: ");**

**Serial.println(mpu.getGyroY());**

**delay(100);**

**}**

We then tested the motors using basic H-bridge code, using the INA and INB pins to switch the direction of current flowing through our motors, and the PWM pins to determine the voltage applied to them. This helped us determine the pin order to use to get the motors to spin in the correct direction. We also tested for the lower bound for speed. With an input to the analogWrite() function of approximately 15, the motors would start to stall and would spin inconsistently. In order to keep our motors safe, the final code included a few lines that would stop the motors when the input dropped below 15.

#define INA1 8 // Direction 1

#define INB1 9 // Direction 2

#define PWM1 10 // Speed PWM

// Motor 2 (Right or B)

#define INA2 13 // Direction 1

#define INB2 12 // Direction 2

#define PWM2 11

void setup() {

Serial.begin(115200);

Serial.println("Starting Motor Test...");

pinMode(INA1, OUTPUT);

pinMode(INB1, OUTPUT);

pinMode(PWM1, OUTPUT);

pinMode(INA2, OUTPUT);

pinMode(INB2, OUTPUT);

pinMode(PWM2, OUTPUT);

}

void loop() {

Serial.println("Forward");

digitalWrite(INA1, HIGH);

digitalWrite(INB1, LOW);

analogWrite(PWM1, 16);

digitalWrite(INA2, LOW);

digitalWrite(INB2, HIGH);

analogWrite(PWM2, 16);

delay(2000);

Serial.println("Reverse");

digitalWrite(INA1, LOW);

digitalWrite(INB1, HIGH);

analogWrite(PWM1, 17);

digitalWrite(INA2, HIGH);

digitalWrite(INB2, LOW);

analogWrite(PWM2, 17);

delay(2000);

Serial.println("Brake");

digitalWrite(INA1, HIGH);

digitalWrite(INB1, HIGH);

analogWrite(PWM1, 0);

digitalWrite(INA2, HIGH);

digitalWrite(INB2, HIGH);

analogWrite(PWM2, 0);

delay(2000);

Serial.println("Coast");

digitalWrite(INA1, LOW);

digitalWrite(INB1, LOW);

analogWrite(PWM1, 0);

digitalWrite(INA2, LOW);

digitalWrite(INB2, LOW);

analogWrite(PWM2, 0);

delay(2000);

}

We then wrote the final PD controller code. We calculated the error during each step by subtracting the current angle from the target angle of 0 degrees. The proportional component could then be determined by multiplying our Kp value by that error. The derivative component was determined by first finding the time change between the current step and the previous step, and dividing that into the change in the error value between the current and previous step. This was then multiplied by the Kd value. The total motor input was found by adding these two components together, and this was sent to the PWM pins to determine speed.

We also included a safety stop into the program. The main code loop was placed within an eternal while loop, and we added a check at the end of each step to ask if the current angle had exceeded our safe limit of 25 degrees. If it had exceeded that limit, the code would break out of the main loop and stop everything.

#include <Wire.h>

#include <MPU6050\_light.h>

MPU6050 mpu(Wire);

float prev = 0;

float target = 0;

float kP = 3;

float kD = 450 ;

unsigned long t;

float lowBound = 15;

#define L1 8

#define L2 9

#define R1 12

#define R2 13

#define LS 10

#define RS 11

void setup() {

Serial.begin(9600);

//pinmodes

pinMode(L1, OUTPUT);

pinMode(L2, OUTPUT);

pinMode(R1, OUTPUT);

pinMode(R2, OUTPUT);

pinMode(LS, OUTPUT);

pinMode(RS, OUTPUT);

Wire.begin(); // On Uno: SDA = A4, SCL = A5

byte status = mpu.begin();

Serial.print("MPU6050 status: ");

Serial.println(status);

while (status != 0) {

delay(1000);

Serial.println("MPU not connected...");

}

Serial.println("Calibrating gyro...");

delay(1000);

mpu.calcOffsets(); // gyro and accel calibration

Serial.println("Done!");

}

void loop() {

//mpu.update();

while(true){

mpu.update();

float current = float(mpu.getAngleX()); //read mpu

int timestep = millis()-t;

t = millis();

//error

float error = target-current;

//P

float P = error \* kP;

//D

float D = kD \* (error-prev)/timestep;

//overall

float motorInput = P + D ;//later

Serial.print(t);

Serial.print(",");

Serial.print (error);

Serial.print(",");

Serial.print (current) ;

Serial.print(",");

Serial.print (P) ;

Serial.print(",");

Serial.print (D) ;

Serial.print(",");

Serial.println (motorInput );

if(motorInput >= lowBound){

motorInput = constrain(motorInput,lowBound,255);

motorMove(motorInput,false);

}else if (motorInput <= -lowBound){

motorInput = constrain(motorInput,-255,-lowBound);

motorMove(abs(motorInput),true);

}else{

motorInput = 0;

motorStop();

}

prev = error; //for next cycle

if (abs(current) >= 25){

break;

}

}

motorStop();

while(true){}

}

void motorMove(int input, bool dir){

if(dir){ //forward

digitalWrite(L1, HIGH);

digitalWrite(L2, LOW);

digitalWrite(R1, HIGH);

digitalWrite(R2, LOW);

}else{

digitalWrite(L1, LOW);

digitalWrite(L2, HIGH);

digitalWrite(R1, LOW);

digitalWrite(R2, HIGH);

}

analogWrite(LS, input);

analogWrite(RS, input);

}

void motorStop(){

digitalWrite(L1, LOW);

digitalWrite(L2, LOW);

digitalWrite(R1, LOW);

digitalWrite(R2, LOW);

}

**Results**

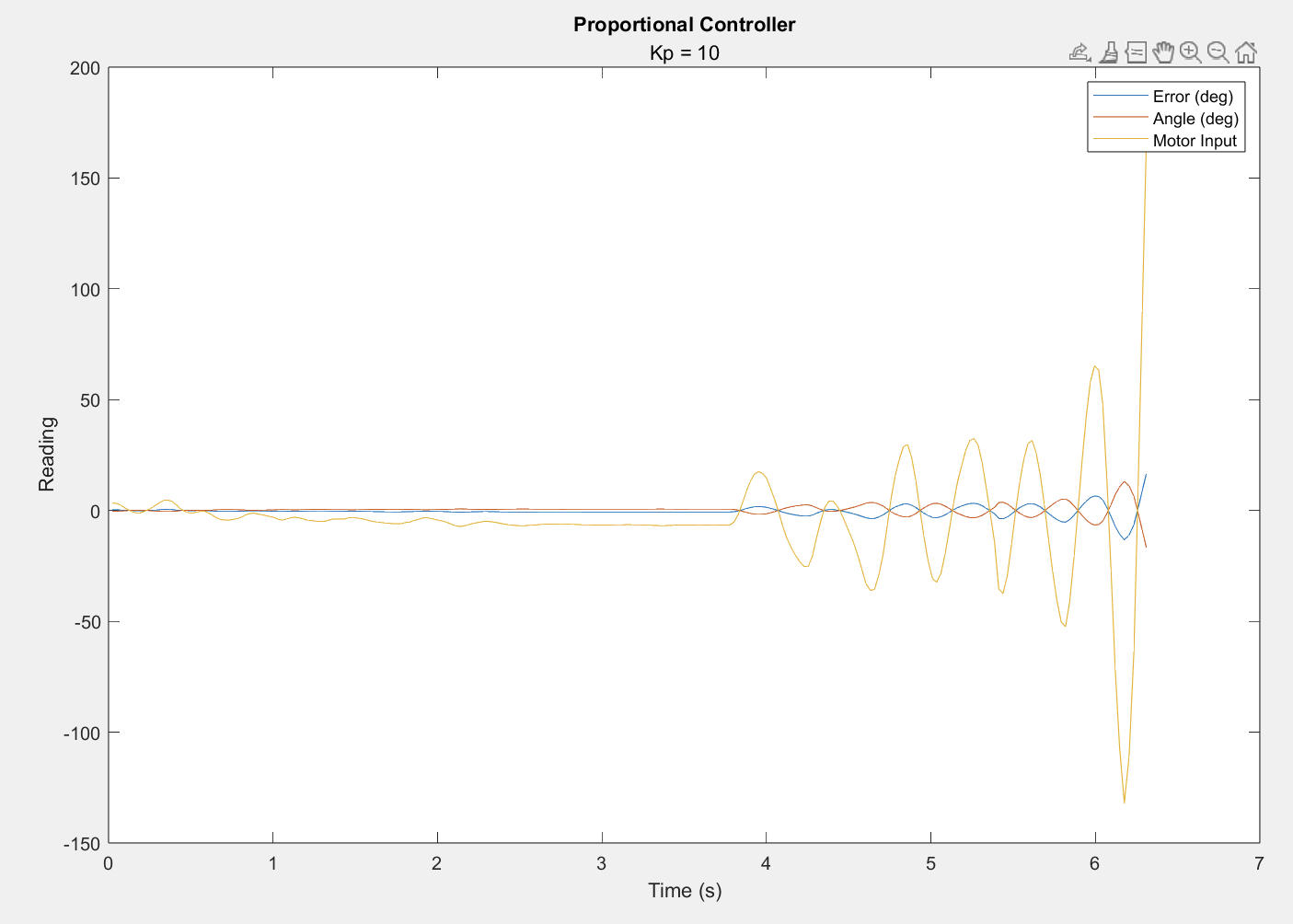
**Data Acquisition**

We gathered data by using the MS Data Streamer add-on for Excel, which reads the serial port similarly to the Arduino Serial Monitor, and can record the printed data into a .csv file. We recorded values for the time, angle error, current angle, the proportional and derivative components, and the total motor input. The motor input and PD terms were the values inputted into the analogWrite() command, with 255 as the maximum mapping to a 12V motor input.

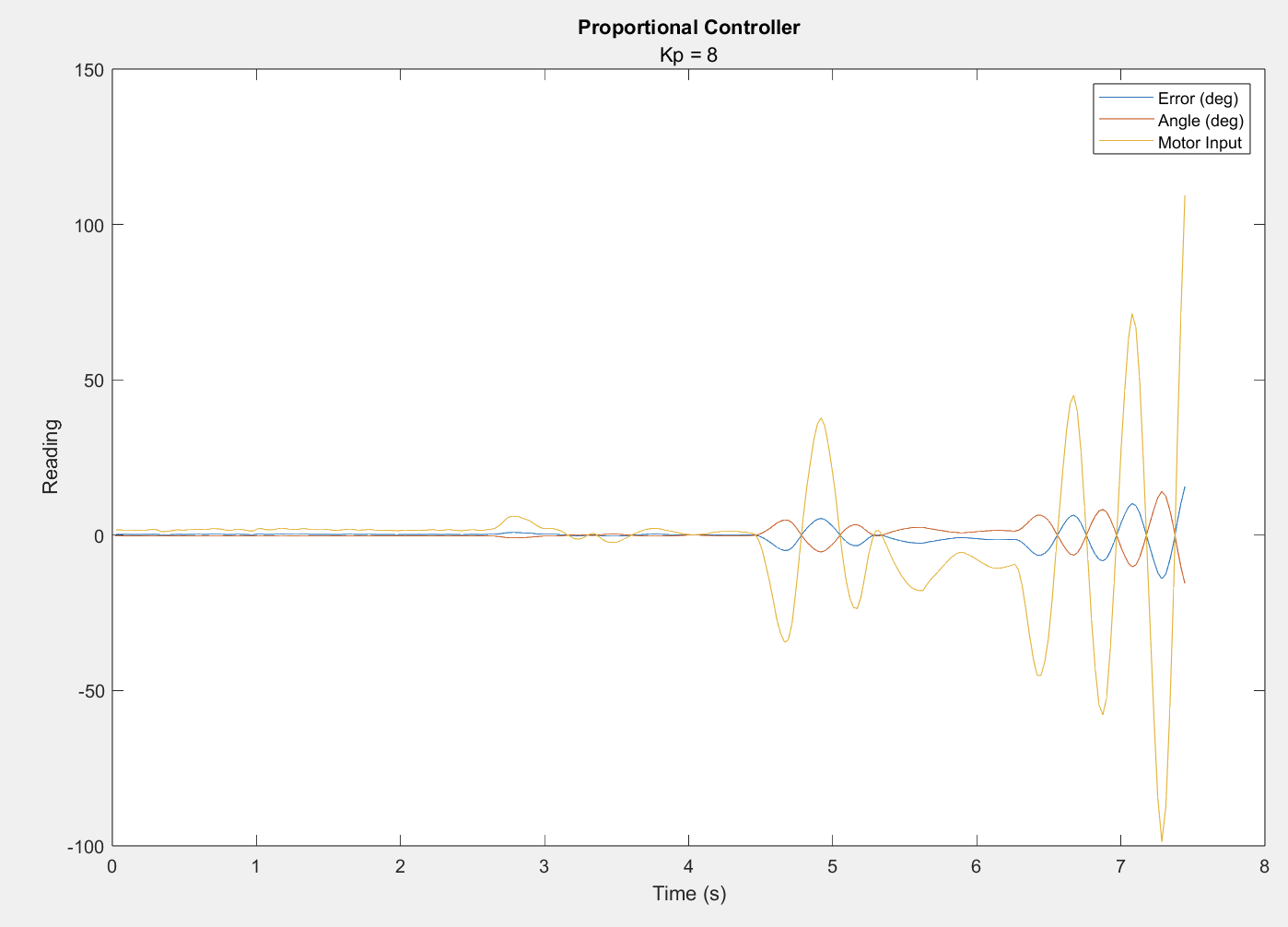
Initial guesses for Kp and Kd were tested by running the robot without power going to the motors, allowing us to read the values the Arduino was printing while we manually turned the robot. This let us ensure that those values made sense and would result in safe operation once power was turned on.

**Proportional Controller**

We began by testing with just the proportional controller by simply removing the line of code that added the P and D components together, and just using the P component as the motor input.

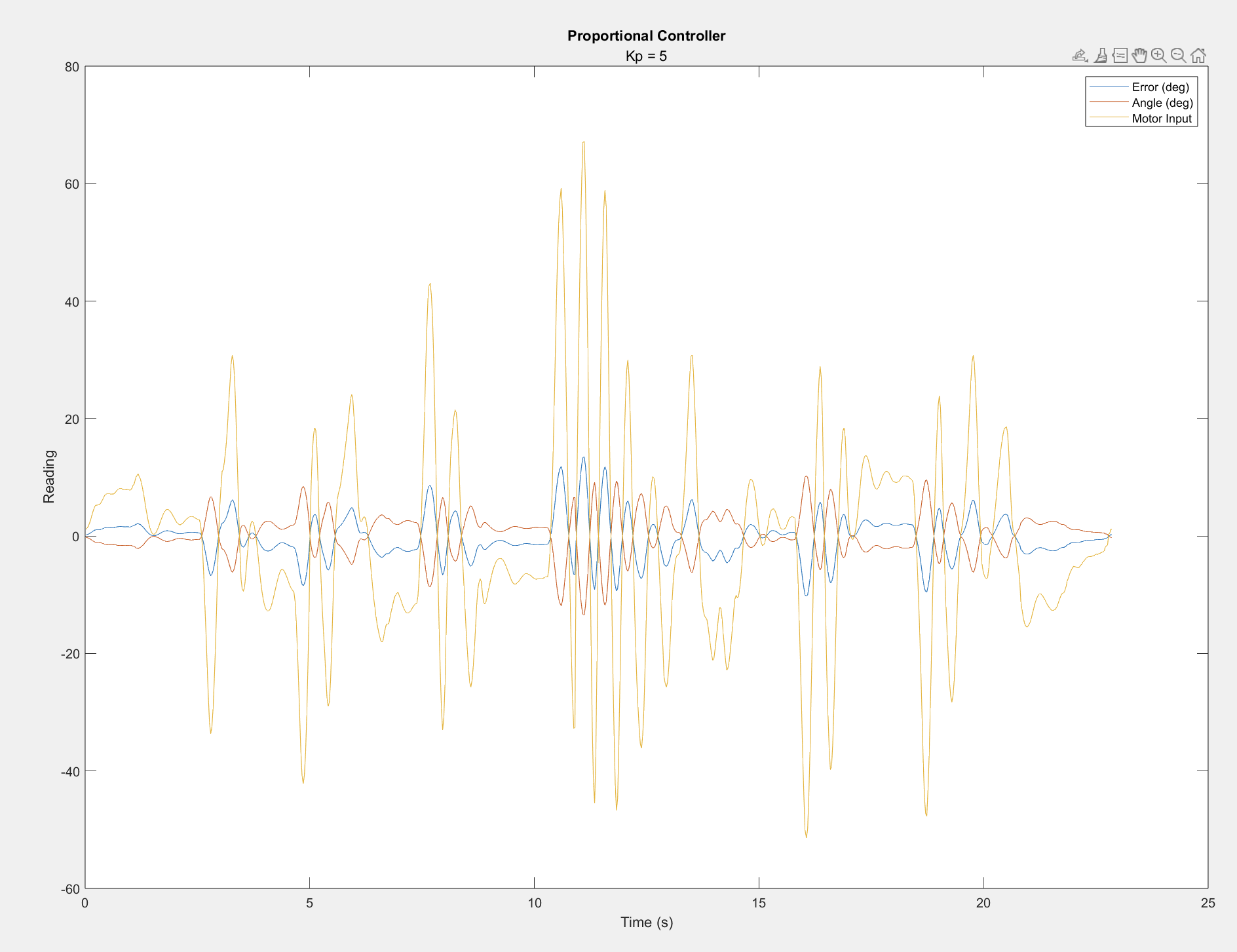


*Fig 5: Proportional controller with Kp = 10*



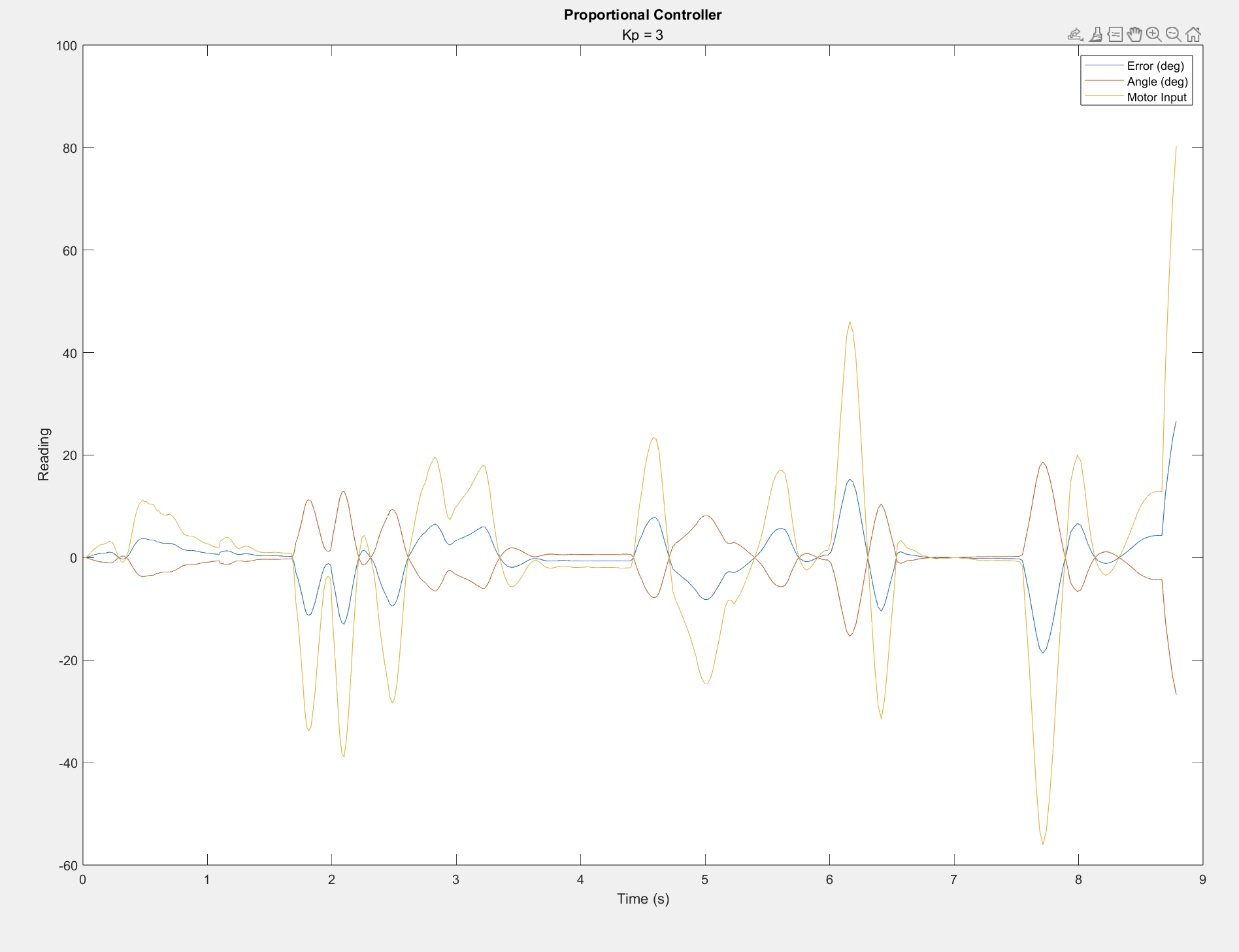
*Fig 6: Proportional controller with Kp = 8*

With higher values for Kp, the robot was unstable and experienced growing oscillations that resulted in it exceeding the safe angle limit we set.



*Fig 7: Proportional controller with Kp = 5*

With a Kp of 5, the robot was stable to small disturbances. Larger disturbances caused it to oscillate a lot, as seen at about 10s in **Fig #**, but it eventually settled back down again.

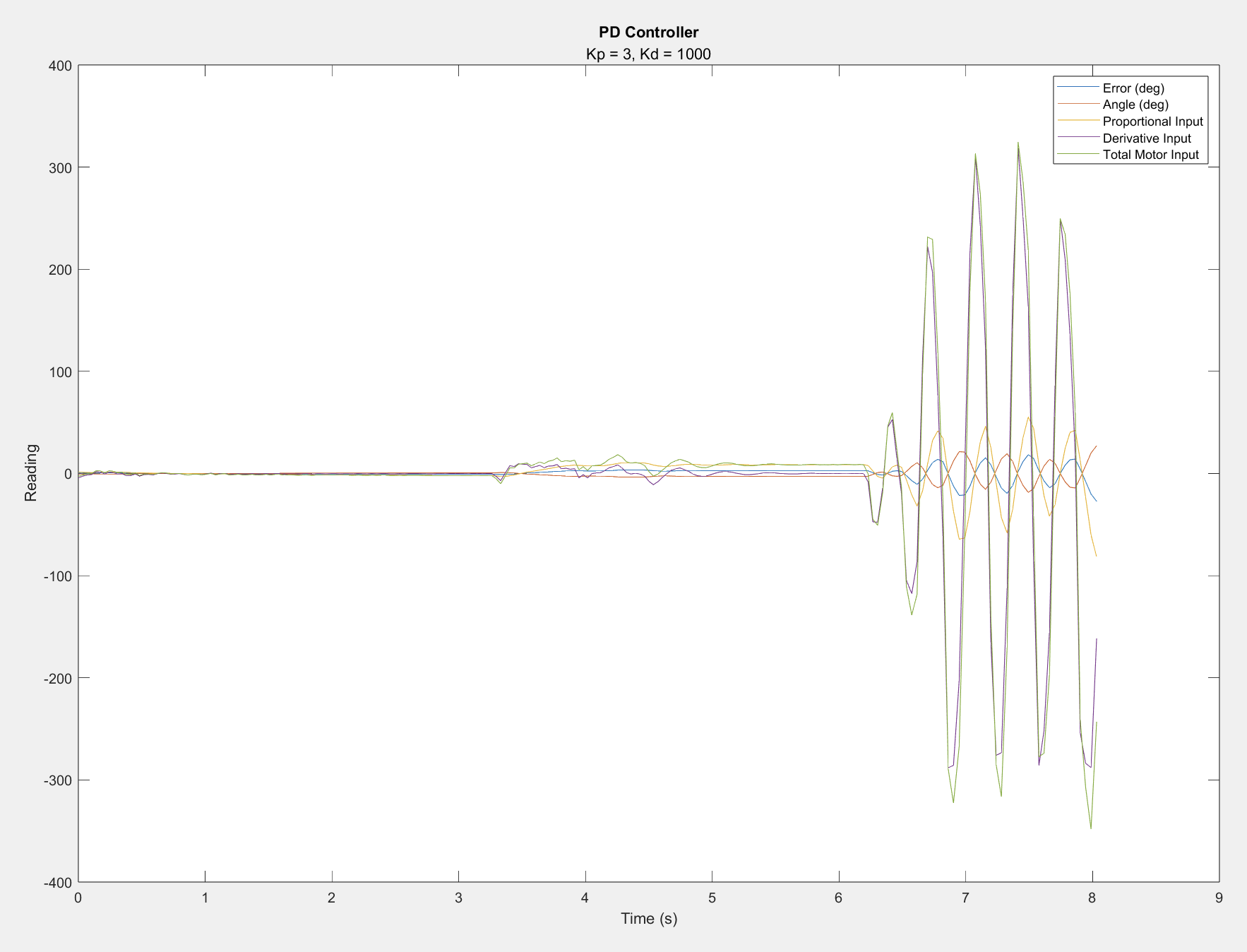


*Fig 8: Proportional controller with Kp = 3*

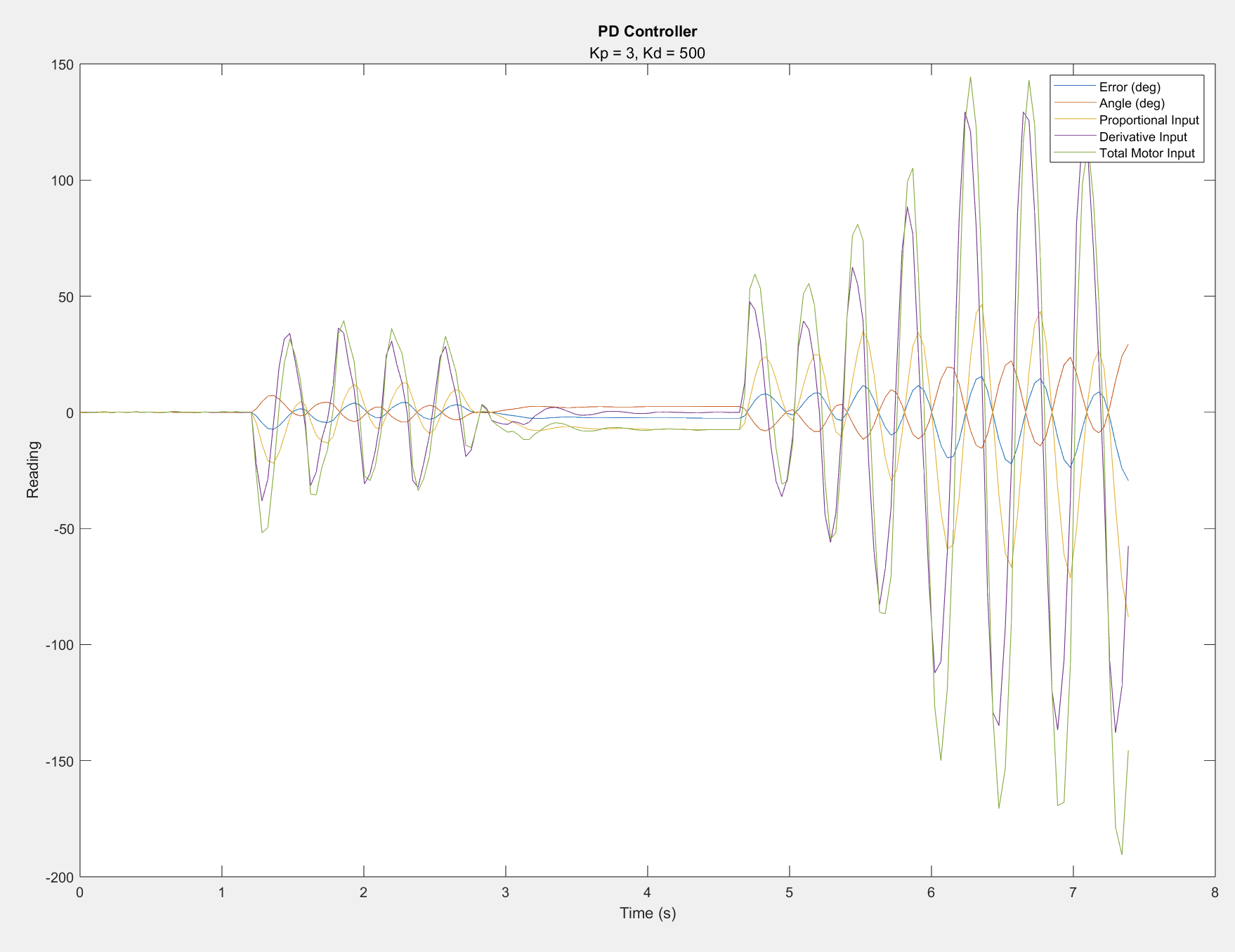
With a Kp value of 3, the oscillations were gone and the controller was stable. The settling time is dependent on the magnitude of the disturbance, but was around 2 seconds.

**Proportional-Derivative Controller**

To try and speed up the response, we added in the derivative term to make a PD controller.

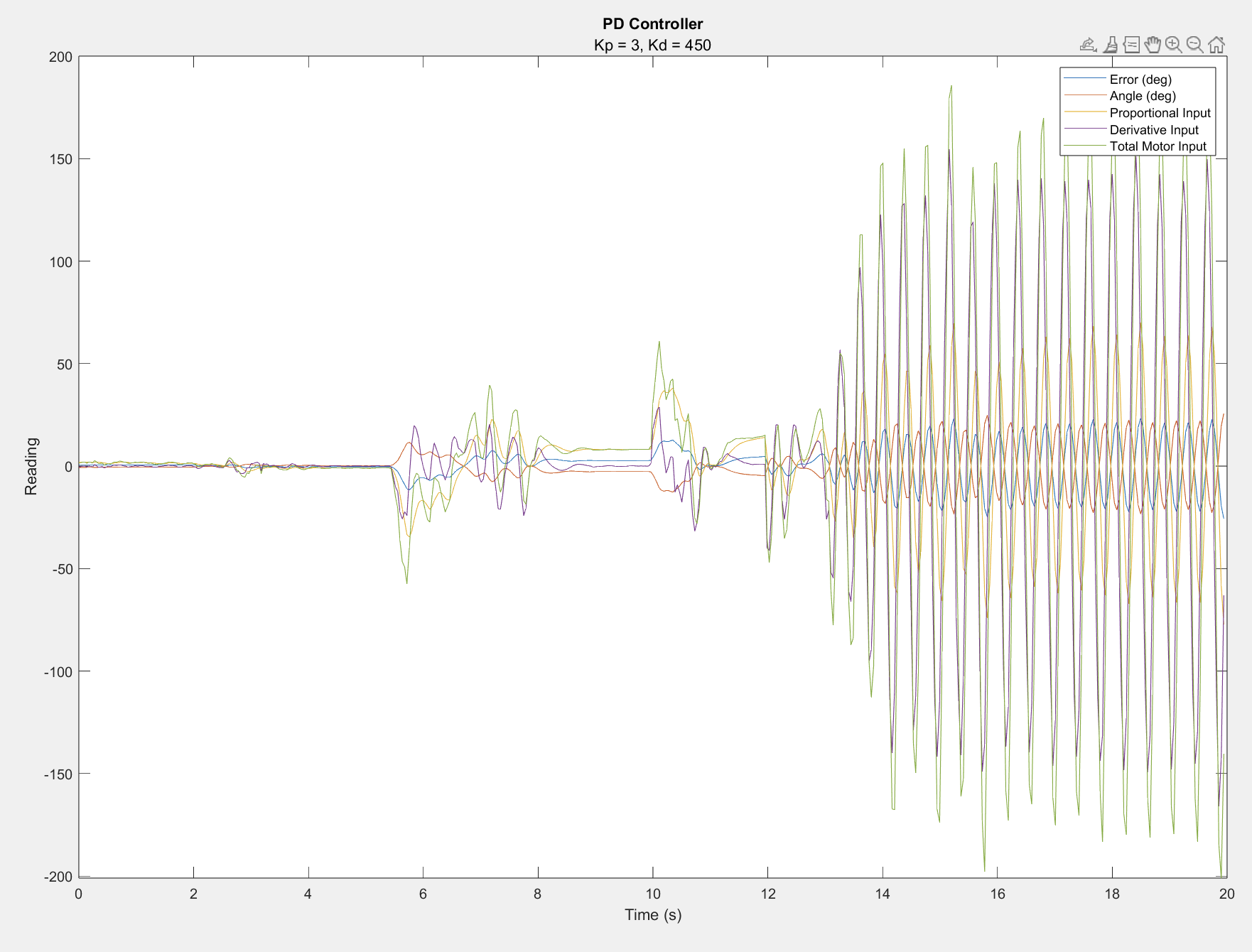


*Fig 9: PD Controller with Kp = 3, Kd = 1000*



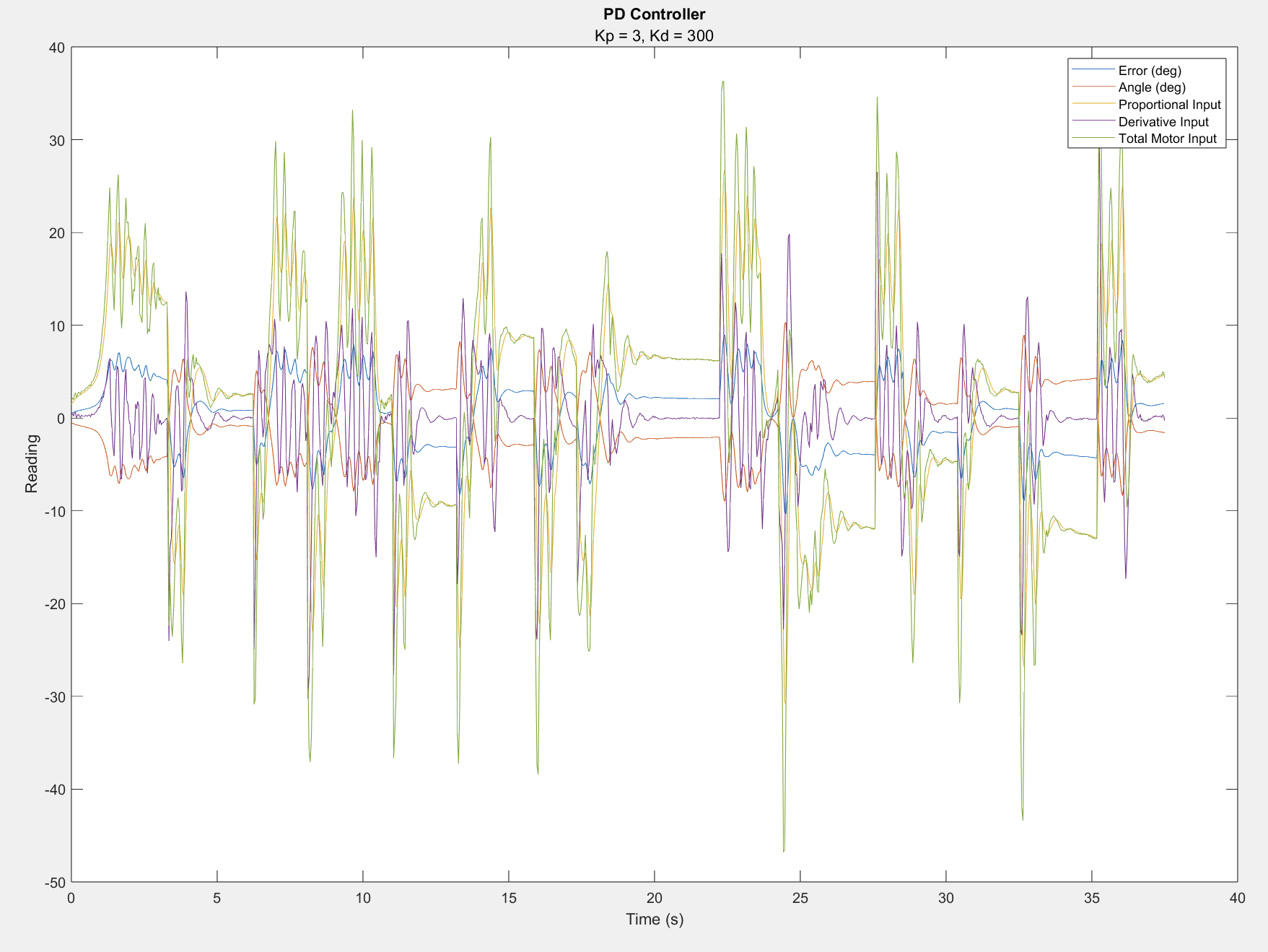
*Fig 10: PD Controller with Kp = 3, Kd = 500*

Similar to the proportional controller, higher values of Kd cause the robot to destabilize when disturbances are applied.



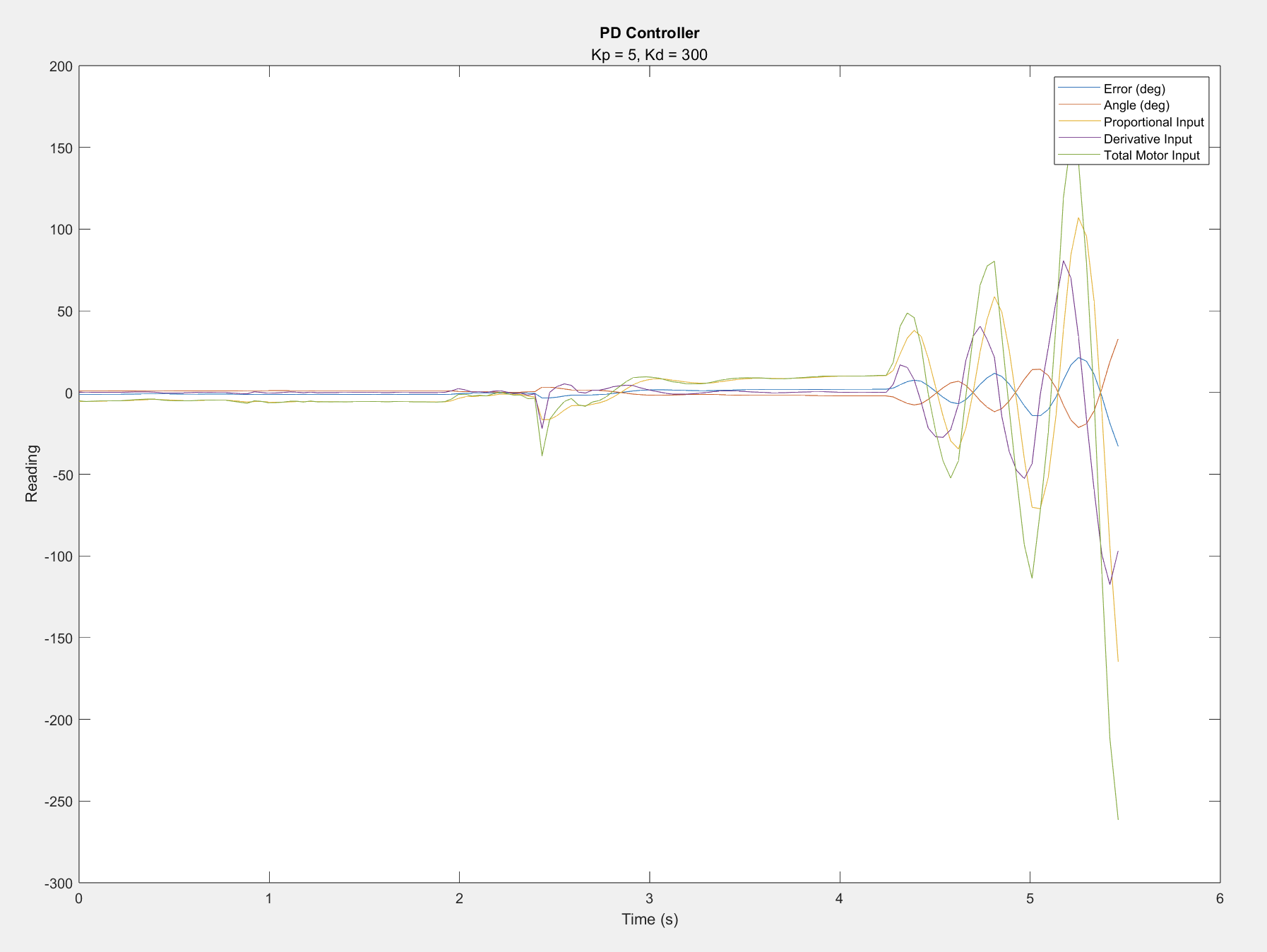
*Fig 11: PD Controller with Kp = 3, Kd = 450*

A Kd value of 4.5 caused the robot to be stable for very small disturbances, but is only marginally stable for larger disturbances, which caused it to oscillate continuously without decaying or stabilizing.



*Fig 12: PD Controller with Kp = 3, Kd = 300*

A Kd value of 3 kept the robot stable, and responded to disturbances slightly faster than the proportional controller, with a settling time of around 1.5 seconds. It also helped it travel less distance while stabilizing, which can be seen in some of the videos linked in the appendix.



*Fig 13: PD Controller with Kp = 5, Kd = 300*

We also tested to see if the derivative term would help stabilize the controller for higher values of Kp, but the robot was still unstable.

**Adding Extra Weight**

Once the robot successfully balanced itself, we tested with an extra weight of approximately 0.3kg on the top of the stick. We were unable to find a combination of Kp and Kd such that the robot was able to rebalance itself back to vertical, but it also did not fall over completely. Instead, it would move in the direction the stick was pushed, acting similarly to a Segway. We believe this occurred because the robot could not achieve the speed or torque needed to push the added weight back up, but it did have the strength to keep it at an angle while it was in motion.

**Conclusion**

The robot was successful, though some improvements could have been made. The motors we used were too fast, and did not have enough torque at low speeds to move the robot. This caused some issues where the controller could not get the robot back to the correct angle because the motors could not move. The wheels we had also did not have enough grip, so at high speeds they would slip and the controller would not work properly. However, despite these issues, we found tuning values that worked and the PD controller successfully kept the robot stabilized against disturbances.

**Appendix**

A folder of assorted videos of our robot in action can be found here:

<https://drive.google.com/drive/folders/1PLejTndw4LxtoSVA8v18seIj9uFeChZ4?usp=sharing>