Ekin Gürgen, Michaela Hennebury
Bench 201
March 6, 2020

ELE302 Report 1: Speed Control

Objective

The goal of this first task was to make our car travel at 4ft/sec for 40 feet on flat ground, going up a ramp, and going down a ramp. In order to do this, we had to accomplish two subgoals: obtain a real-time car speed measurement using a Hall Effect sensor and implement PI feedback control to provide a pulse-width modulated signal to our motor.

The components of this project can be organized into three groups: sensors, actuators, and control. The Hall Effect sensor falls in the sensor category, the car's motor is an actuator and the PI feedback loop is our control system. These three units are used in conjunction to accomplish the speed control task.

Overview of Subsystems and Components

Power Board

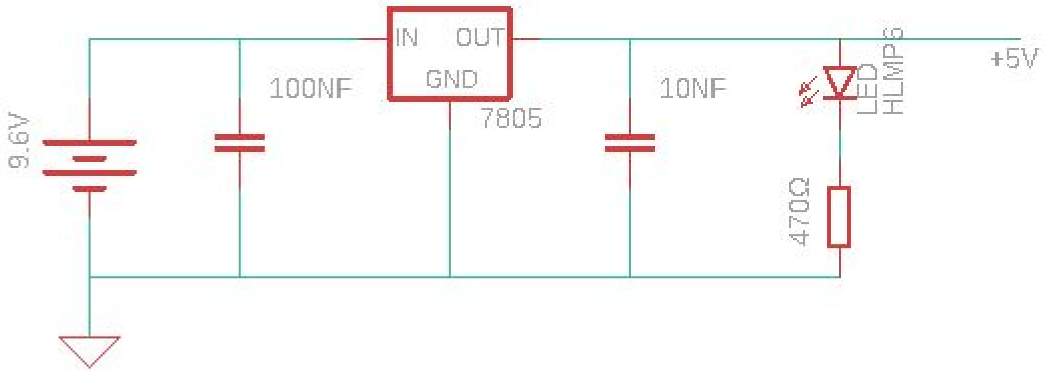


Figure 1: Schematic of the Power Board.

The purpose of the power board is to convert 9.6V from the battery source to 5V to supply the Hall Effect sensor and the PSoC. This is accomplished using the voltage regulator, labeled as 7805 on the circuit schematic. The power board receives an input of 9.6V from the battery, which is connected to the terminal block. This is the input to the voltage regulator, which has an output of 5V that goes to the Hall board and the PSoC. A red LED is put in series with a 470Ω resistor (so that the 5V output is not shorted to ground), and the board is working when the LED is on. The capacitors on either side of the voltage regulator are decoupling capacitors. These serve to reduce noise in the rest of the circuit.

Hall Effect Sensing

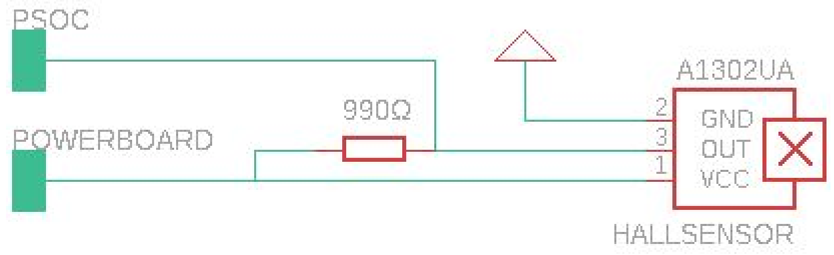


Figure 2: Schematic of the Hall Board.

The Hall Effect sensor takes 5V from the power board. The wheel of the car has five magnets inside the rim, and the sensor is glued to the chassis such that the magnets pass directly by it when the wheel is turning. The output of the sensor is 5V (the same as the input) when the magnet is not above the sensor, and is 0V when the magnet is above the sensor. We verified that this component was working correctly with the oscilloscope. In our PSoC code, we scheduled an interrupt when there was a rising edge in the sensor signal, allowing the program to record the time in between magnets on the wheel and calculate the current speed of the car.

Motor Board

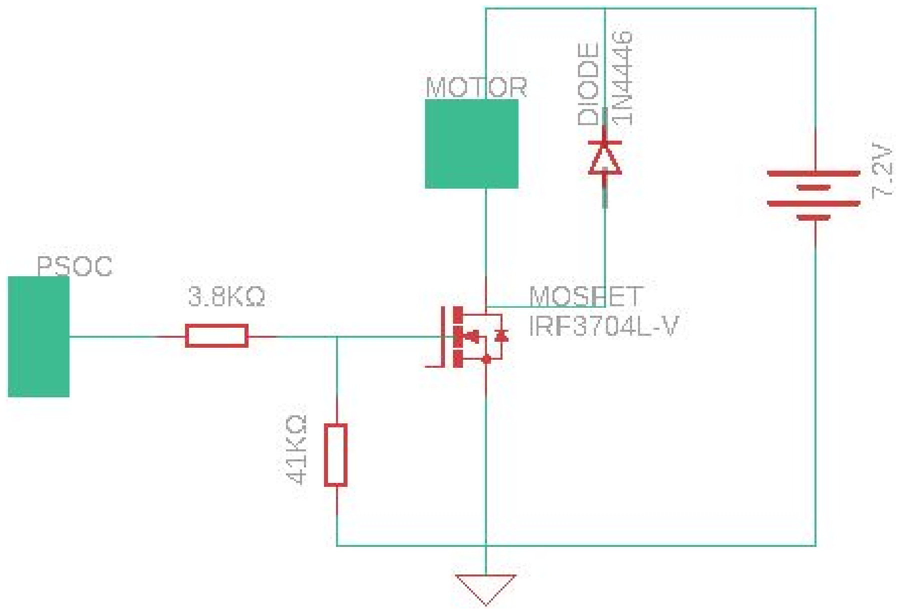


Figure 3: Schematic of the Motor Board.

The power board takes a pulse-modulated (amplitude is 5V) input from the PSoC. This signal serves as the gate voltage for a MOSFET. When the gate voltage is 0V and the FET is off, no current is allowed to pass through the motor. When the gate voltage is high, current passes through the motor. A flyback diode is attached in parallel with the motor to protect it from high current.

How did we choose the MOSFET? We used an IRF3708 device. The two main factors in choosing this device were making sure that the threshold voltage and the on-current were appropriate for our setup. The threshold voltage of this device is 0.6-2V. The signal from the PSoC is ~5V, which will turn the FET on easily. We also looked at the I-V characteristics of the FET (shown in Figure 4 below) to make sure the current was high enough with our V_{GS} and V_{DS} parameters. At V_{GS} = 5V and V_{DS} = 7V, the on-current is >100A, which is more than sufficient to turn the motor on.

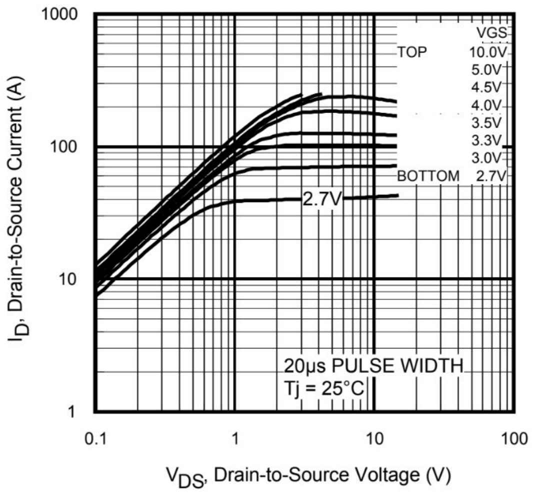


Figure 4: Typical IRF3708 output characteristics.

When we were first testing the motor board, we discovered an issue where the motor would stay powered regardless of the FET gate voltage. This was because the voltage difference between the gate and the source was floating, causing current to flow at all times. To fix this, we added a 41kΩ resistor from the gate to the source (grounded). This resistor also created a voltage divider between the PSoC input and the FET gate. To ensure that this would not cause the input signal to be lower than the gate voltage, we chose resistance that was ~10x larger than the resistor protecting the FET.

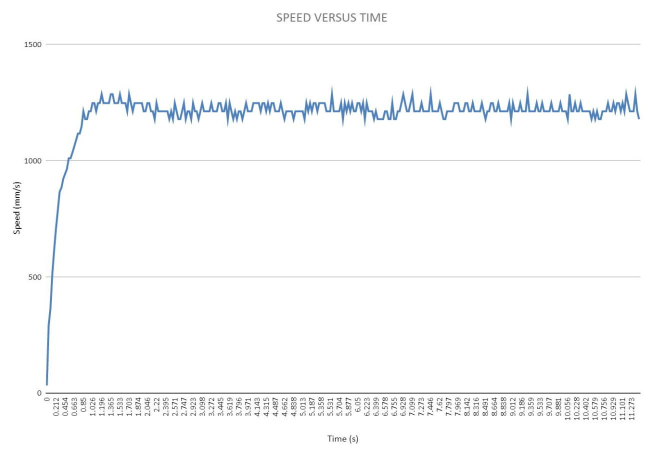
PSoC

The PSoC functions as the brain of our car, capturing the time in between signals from the Hall board, running our PI control program and outputting an appropriate pulse-width modulated signal to the motor board. More details on PI control are included in the following section. We also used the XBee on the PSoC as a means to debug and collect data during our building and testing process.

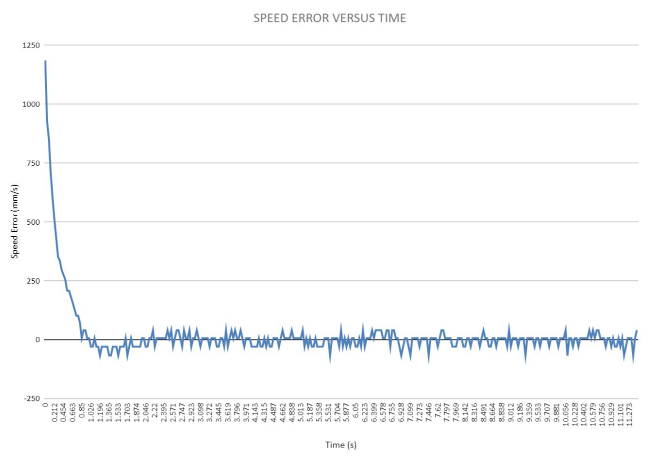
Results

After writing the code that converts sensor signals to speed, our goal was to design a PI controller that can keep the car travelling at a reference speed of 122 cm/s (4 ft/s). A heuristic approach known as the Ziegler-Nichols method was used to tune the parameters of the PI controller. Using this method, we initially set the integral gain K_I to 0 and increased K_P until the car's speed had stable and consistent oscillations. This oscillatory behavior was observed around $K_P = 0.09$. Then, we set K_P to 0.05 for a quarter amplitude decay behavior and started increasing K_I for achieving zero steady-state error. We found that $K_I = 0.005$ was a good value where the integral term was large enough for any steady-state error to be corrected and still small enough to not affect the stability of our systems badly. A Python script (provided in the Appendix) was used to plot the "Speed versus Time" and "Speed Error versus Time" graphs given below. As it can be seen in Graphs 1, 3 and 5, using the good K_P and K_I values, we were able to reach the reference speed without too much overshoot and stabilize around it while going on a flat surface as well as going up and down a ramp. In addition, Graphs 2, 4 and 6 show the Speed Error which stabilizes around 0 for all three cases.

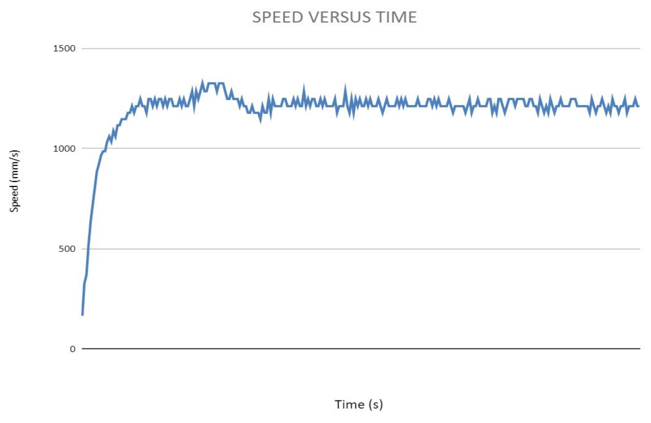
Graph 1: Speed vs Time (Good PI, Flat)



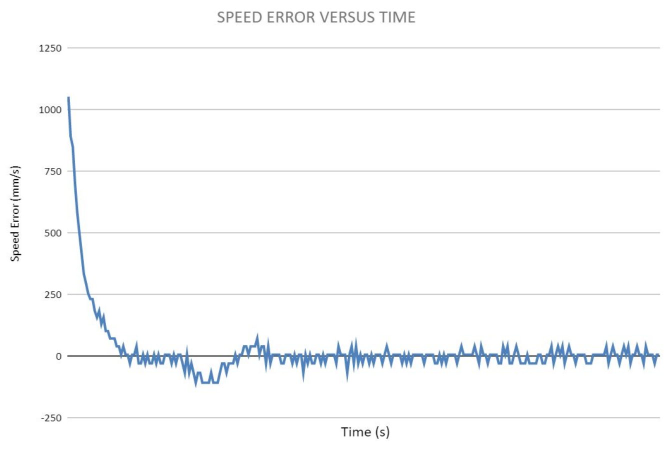
Graph 2: Error vs Time (Good PI, Flat)



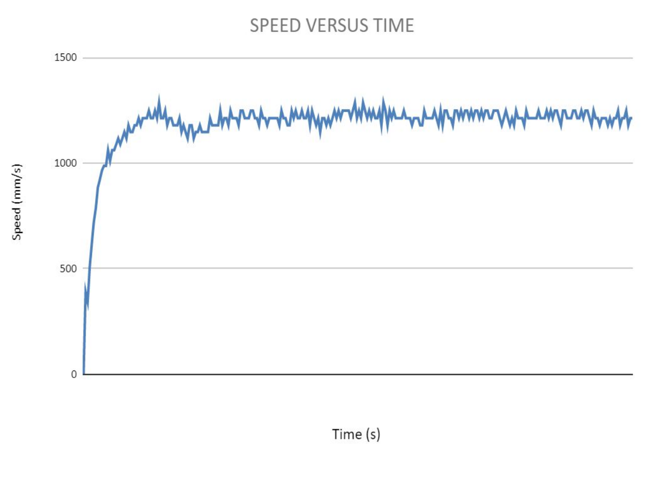
Graph 3: Speed vs Time (Good PI, Down)



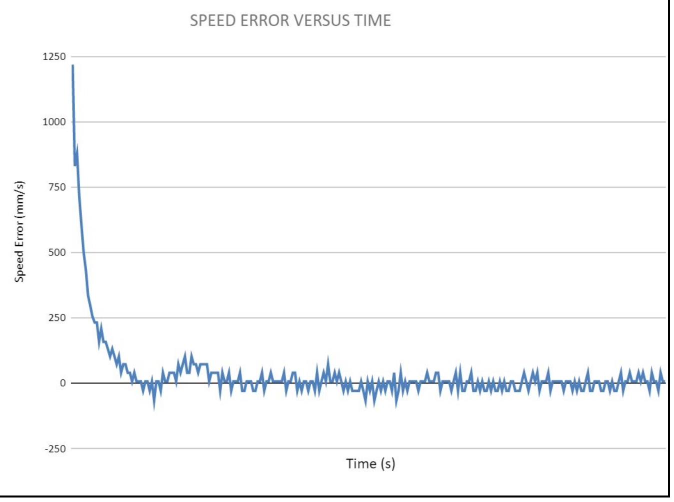
Graph 4: Error vs Time (Good PI, Down)



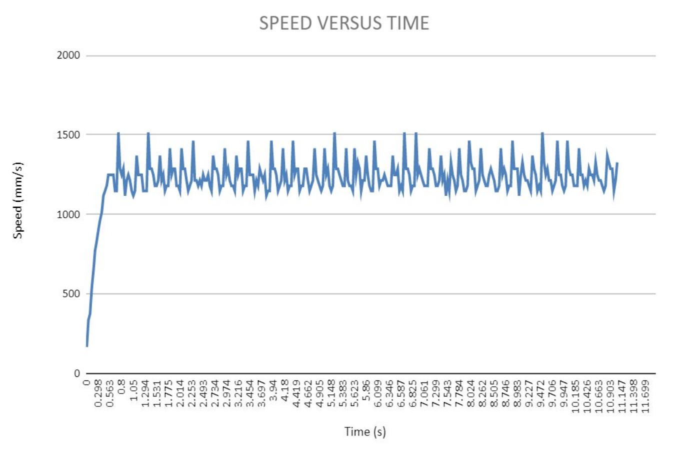
Graph 5: Speed vs Time (Good PI, Up)



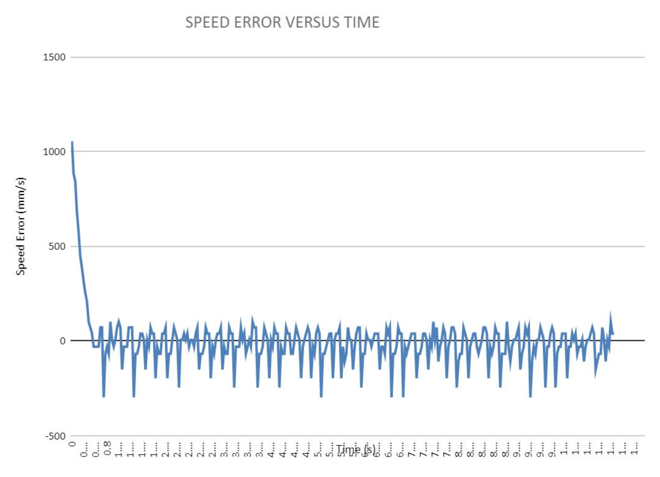
Graph 6: Error vs Time (Good PI, Up)



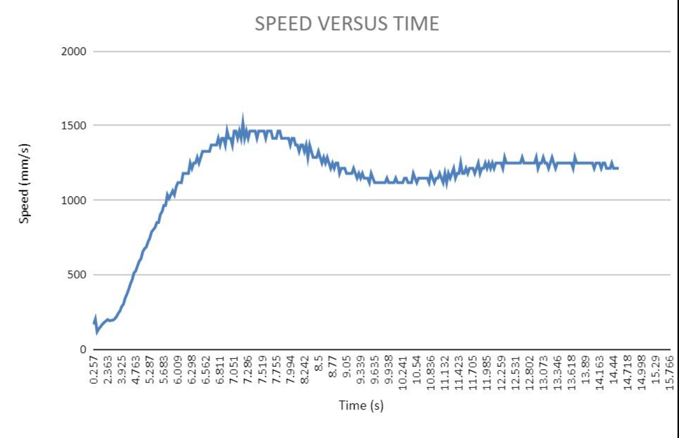
Graph 7: Speed vs Time (High Gain Control)



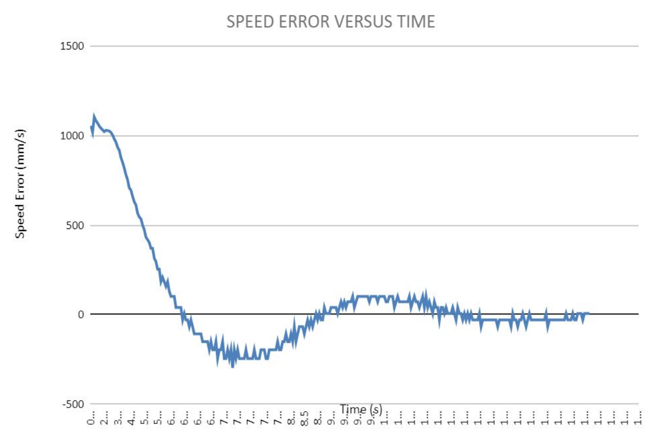
Graph 8: Error vs Time (High Gain Control)



Graph 9: Speed vs Time (Low Gain Control)



Graph 10: Error vs Time (Low Gain Control)



To improve our understanding of the PI control, we experimented with different parameters and plotted the results in Graphs 7 through 10. Using $K_P = 0.5$ and $K_I = 0.005$, we gathered Graphs 7 and 8, where the proportional gain was too high, which slightly decreased the rise time and caused an aggressive oscillatory behavior which can easily be differentiated from noise by the high amplitude. Using $K_P = 0.005$ and $K_I = 0.0005$, we gathered Graphs 9 and 10, where the gain being too low substantially increased the rise time from being under 1 second to around 6 seconds and increased the overshoot. On contrary to the high gain example, having low gain values decreased the controller's sensitivity to external disturbances and made it very slow at reference tracking.

Discussion

Overall, progress on this project was relatively smooth. We had no significant issues, but there are a few things that we would like to approach differently in the future:

1. We need to start being more rigorous when testing components and circuitry. For some of the boards, we simply grabbed components from the drawers, soldering everything together and powered it afterwards to make sure everything was working. We were lucky this time, but in the future we should test individual components (LEDs, diodes, etc.) and construct our circuits on a non-permanent breadboard to ensure that everything is working correctly before soldering.
2. Most of the issues that we did have were due to poor connections. At different points, we had issues with soldered connections, crimp connections and KK connections, and while these are easy to fix they proved difficult to find (for instance, I thought something had broken on our power board when it really turned out to be a broken crimp connection on the terminal block). Now that we have slightly more experience with building these connections, we will hopefully have fewer problems with this in later projects, but it would be useful to develop better strategies for debugging problems with the circuitry.
3. While we did not have many challenges with the code, we should use more conventional units in the future (writing everything in cm/100sec got confusing late at night). It was a good idea to use both the LCD screen and the XBee to ensure that the PSoC was programming properly and using the desired information.

APPENDIX

Source Code (main.c)

```
#include "project.h"
#include <stdio.h>

int error; // present error
int accError = 0; // accumulated error for integral control
int diffError = 0; // differential error for derivative control
int prevError = 0; // previous error
int refSpeed = 12190; // mm/sec
int time; // time since last Hall sensor reading
int PWMnow = 40; // pulse width modulation to motor between 0 and 255, 40 is the initial
value
int startCount = 0; // start integral control after certain time


CY_ISR(inter) { // runs when Hall sensor reads
   int captureVal;
   char strbuf[64];
   double speed;

   startCount++;

   captureVal = Timer_ReadCapture();
   time = (65536 - captureVal); // milliseconds
   speed = (3.93 / time)*1000; // mm/sec
   speed *= 1.08; // correction factor, 3.93 may not be completely accurate

   int speed100cm = (int) (speed * 100);
   if (startCount > 10) { // start integral control later, not immediately
      accError += error;
   }
   prevError = error;
   error = refSpeed - speed100cm; // in cm/sec
   diffError = (error - prevError) / (time*1000);

   sprintf(strbuf,"Time: %i Speed: %i Error: %i PWMnow: %i", time, speed100cm, error,
PWMnow); // time in ms, speed in mm per sec

   // print to UART for debugging and collecting data
   char strbuf2[64];
   sprintf(strbuf2,"%s\r\n",strbuf);
   UART_PutString(strbuf2);
```

```
}

int PIController() {
    double kp = 0.05; // proportional constant
    double ki = 0.005; // integral constant
    double kd = 0.0; // differential constant

    double P_term = 0;
    double I_term = 0;
    double D_term = 0;

    // Compute the P term
    P_term = kp * error;

    // Compute the I term
    if (startCount > 10) {
        I_term = ki * accError;
    }

    // Compute the D term
    D_term = kd * diffError;

    int result = P_term + I_term + D_term;


    if (result > 255){ // make sure result is within range for car to work
        result = 255;
    }
    else if (result < 0) {
        result = 0; // arbitrary, may change
    }

    return result;
}

void UpdatePWM(int fOutput) { // change the pulse width modulation to motor

    PWMnow = fOutput;
    PWM_WriteCompare(PWMnow);

}

int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    int fOutput;


    /* initialization code */
    PWM_Start();
    UART_Start();
    LCD_Start();
    Timer_Start();

    LCD_Position(0,0);
    LCD_PrintString("C"); // make sure correct code got to PSoC :)

    PWM_WriteCompare(PWMnow);
    error = refSpeed; // To TEST initial conditions

    Hall_Interrupt_Start();
    Hall_Interrupt_SetVector(inter);

    for(;;)
    {
        // continuously update the motor
        fOutput = PIController();
        UpdatePWM(fOutput);

    }
}

/* [] END OF FILE */
```

Code for Plotting the Data (car_data_plot.py)

```
import serial

import time

import matplotlib

import matplotlib.pyplot as plt
import datetime

import xlsxwriter


# create empty lists and create the empty Excel workbook

buffer = []

errorData = []

speedData = []

timeData = []

workbook = xlsxwriter.Workbook('data_ramp_flat.xlsx')

worksheet = workbook.add_worksheet()


COM_PORT = "COM3"

s = serial.Serial(COM_PORT, baudrate=115200, timeout=1)


lines = ""

# sample for 20 seconds

t0 = time.time()

TOTAL_SAMPLE_TIME_SECS = 20

print(f"Sampling for {TOTAL_SAMPLE_TIME_SECS}")

while (time.time() - t0) < TOTAL_SAMPLE_TIME_SECS:

        lines = s.readline().decode().rstrip()

        if len(lines):

                print(lines)
                count = 0
                for word in lines.split():
                    if (count == 1):
                        t = int(word)
                        timeData.append(t)
                    if (count == 3):
                        speed = int(word)
                        speedData.append(speed)
                    if (count == 5):
                        error = int(word)
                        errorData.append(error)
                    count += 1
                buffer.append(lines)
# Save the data to an Excel file
row = 0
column_error = 0
column_speed = 1
column_time = 2

for value in errorData:
    worksheet.write(row, column_error, value)
    row += 1

row = 0
```

```
for value in speedData:
        worksheet.write(row, column_speed, value)
        row += 1


row = 0
for value in timeData:
        worksheet.write(row, column_time, value)
        row += 1


workbook.close()


# Save a matplot of the data (for a quicker access than the Excel file)
x = [datetime.datetime.now() + datetime.timedelta(seconds=i) for i in range(len(errorData))]
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, speedData)
fig.savefig('testfig.png')
```