Faculty of Engineering

School of Electrical, Electronic & Computer Engineering

EERI 418

Practical 1

Completed By

MJ Bezuidenhout 24162299

Submitted to:

Prof. K. Uren

June 5, 2017

DECLARATION

I, MJ Bezuidenhout, declare that this report is a presentation of my own original work. Whenever contributions of others are involved, every effort was made to indicate this clearly, with due reference to the literature. No part of this work has been submitted in the past, or is being submitted, for a degree or examination at any other university or course.

Potchefstroom, June 5, 2017

[illegible]

MJ Bezuidenhout, June 5,
2017

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

PID	Proportional,Integral & differential Controller
PI	Proportional& differential Controller
Ts	Settling Time
PO	Percent Overshoot

INTRODUCTION AND PROBLEM STATEMENT

The purpose of this assignment is to design, simulate and implement a PI controller to compensate a DC motor using a microcontroller. The implementation will be assessed by means of a report in accordance with ECSA ELO 2, as part as the following milestones:

MILESTONE 1 Characterisation of a DC motor

MILESTONE 2 Mathematical modelling and simulation of the DC motor

MILESTONE 3 Controller design and simulation of closed loop system

MILESTONE 4 Practical demonstration of controller design

MILESTONE 5 Final report

The first milestone was completed successfully earlier in the semester and the lab report is included in appendix A.

LITERATURE STUDY

2.1 MODELING THE DC MOTOR

According to the Prescribed textbook[1, p.160]:

> *Time constant*
> The time interval necessary for a system to change from one state to another by a specified percentage. For a first order system, the time constant is the time it takes the output to manifest a 63.2% change due to a step input.

2.2 SIMULATONG THE SYSTEM

> Simulink, developed by The MathWorks, is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. With Simulink, you build models by dragging and dropping blocks from the library browser onto the graphical editor and connecting them with lines that establish mathematical relationships between blocks. You can set up simulation parameters by double clicking the blocks.
> [2, p.4]

MODELING

In this section the characteristics of the DC motor need to be interpreted in a way that allows an appropriate controller to built around it. The List of all the measured parameters, and methods are included in the "One page Experiment design" included in Appendix A.Since the specification only requires a first order model of the motor, and since the mechanical time constant overwhelms the electrical time constant, the mechanical time constant is the only parameter necessary to meet the specifications.

3.1 MEASURING THE MECHANICAL TIME CONSTANT

With regards to the definition of a time constant as discussed in the Literature study, the time constant of the DC motor was calculated by plotting the rotational velocity of the over time after shutting off the power source. Refer to Figure 1 These Measurements were taken by connecting the Tachometer to the PicoScope and recording the values into a .csv file. The Data was prepared for plotting in the following ways:

- The time stamps were offset so that the shutoff occurred at $t = 0$
- The tachometer gave a reading 360mV when the speed was 0 RPM, so the Y axis was offset accordingly
- The Data was smoothed using an average filter. See python script in Appendix B

From the representation of the data in Figure 1, it can be measured that the motor took Approximately 0.77 seconds to reach a speed of 440RPM, A value 63% from the steady state value. The mechanical time constant τ_ℓ is given by:

$$\tau_\ell = 0.77s \tag{1}$$

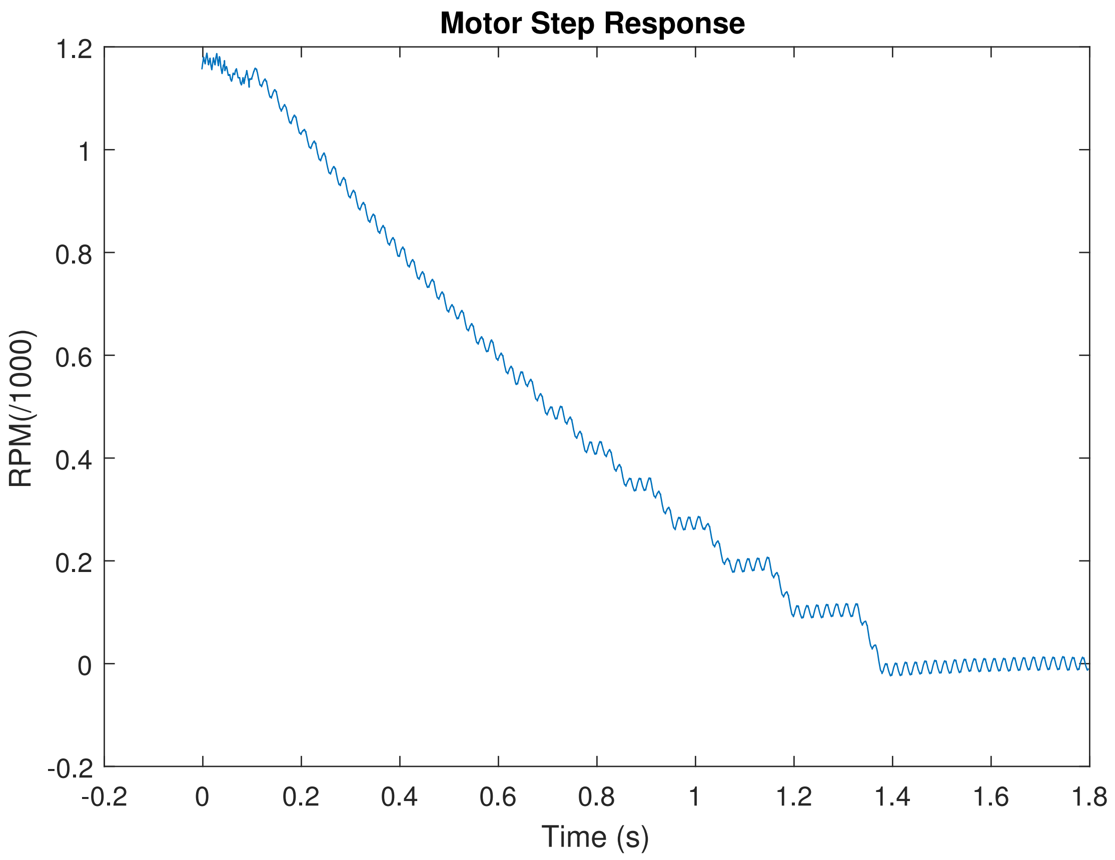


Figure 1: Time Response of the DC motor

3.2 DETERMINING K AND COMPARING THE MODEL WITH THE RESULTS

Having measured the time constant, the time domain model of the DC motor is in the form:

$$\dot{\omega}(t) = Ke^{\frac{t}{\tau_\ell}} \tag{2}$$

Using the value calculated for τ_ℓ and choosing K by means of trial and error, the following model was found to be reasonably close to the measured response:

$$K = 1.2 \tag{3}$$
$$\dot{\omega}(t) = 1.2e^{\frac{t}{0.77}} \tag{4}$$
$$G(s) = \frac{1.2}{0.77s + 1} \tag{5}$$

Refer to Figure 2, where the time domain model is compared to the measured results. It must be noted that, at this stage in the practical process, the motor was modeled as a function of the input voltage, which was in the order of 0-400V. This model, therefore represents the response to a 400V step in input. It was later found that the motor must be compensated using a duty cycle. Finally, both the reference speed and tachometer output are given in the form:

$$1\text{RPM} = 1\text{mV} \tag{6}$$

These considerations obviously render the above transfer function unusable (since a plant input in the order of 100 corresponds to an output in the order of 1) and the gain K, must be adjusted before designing the PI controller. It was found that lowering the gain by a factor of 100 yielded an appropriate plant transfer function. By definition, the time constant stays unchanged, regardless of the gain.The transfer function on which the PI controller will be based is therefore given by:

$$G(s) = \frac{0.012}{0.77s + 1} \tag{7}$$

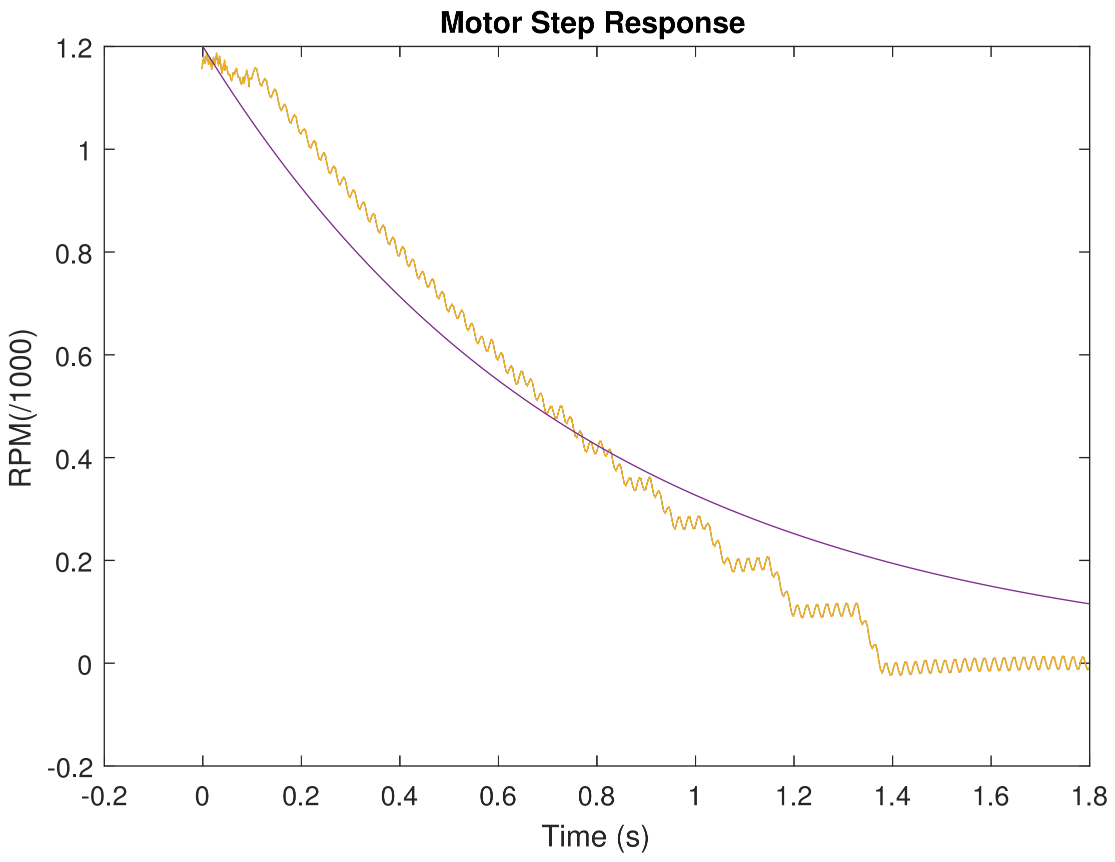


Figure 2: Simulation of the time domain model compared with Laboratory results

3.3 DESIGNING THE PID

The following table shows the specifications that the system must adhere to:

The transfer function for a PI controller in General is given by:

$$G_c(s) = K_c + \frac{K_i}{s} \tag{8}$$

Table 1: Specifications

Parameter	**Maximum Value**
Steady state error	1%
Percent Overshoot(P.O.)	10 %
Settling time	<2s

The equations for Percent overshoot and settling time are given by:

$$P.O. = e^{\left(\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}\right)} \tag{9}$$

$$T_s = \frac{4}{\zeta\omega_n} \tag{10}$$

Furthermore, ζ and ω_n can be used to define a desired characteristic equation in the form:

$$Q(s) = s^2 + 2\zeta\omega_n s + \omega_n^2 \tag{11}$$

The PI controller design can be summed up as the reconciliation of the desired characteristic equation 11 with the characteristic equation of the system, which is given by:

$$Q(s) = G(s)G_c(s) + 1 \tag{12}$$

With respect to the equations 7 and 8
This design is described in the next section.

CONTROLLER DESIGN

4.1 DETERMINING ζ AND ω_n

Substituting the P.O from Table 1 and solving equation 9 yields:

$$\text{P.O.} = e^{\left(\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}\right)} \tag{13}$$

$$0.1 = e^{\left(\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}\right)} \tag{14}$$

$$\zeta = 0.5911 \text{ solved numerically} \tag{15}$$

Substituting the result of Eq 13 and the settling time from Table 1 into eq 10 yields:

$$2 = \frac{4}{\zeta\omega_n}|_{\zeta=0.5911} \tag{16}$$

$$\omega_n = 3.384 \tag{17}$$

4.2 SOLVING FOR K_p AND K_i

Reconciling the compensated transfer function and desired transfer function yields:

$$Q(s) = 1 + G_c(s)G(s) \tag{18}$$

$$Q(s) = s^2 + 2\zeta\omega_n s + \omega_n^2 \tag{19}$$

$$Q(s) = s^2 + s(1.298 + 0.012K_p) + 0.012K_i \tag{20}$$

By the principle of superposition:

$$2\zeta\omega_n = 1.298 + 0.012K_p \tag{21}$$

$$\omega_n^2 = 0.012K_i \tag{22}$$

$$K_p = 255.155 \tag{23}$$

$$K_i = 954.288 \tag{24}$$

$$\tag{25}$$

Note: the values of K_p and K_i seem very high, but this is by design, since the PI controller must have a very high gain, converting an error input in the order of 1V to an output in the order of 100%

4.3 TESTING THE RESULTS

Figure 3 was plotted to confirm the calculations. The result was very close to the specification, in the simulation, these results will be tuned, since the specifications are given as upper limits.

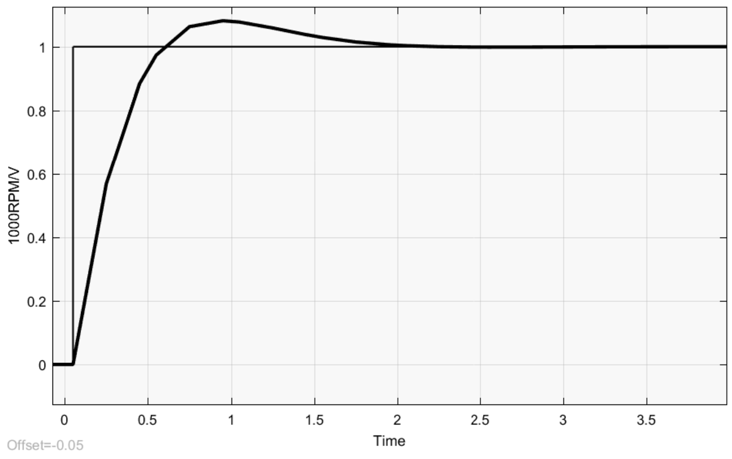


Figure 3: Plot of the transient response of the Caculated system

Table 2 Compares the specification to the theoretical result of the calculated parameters:

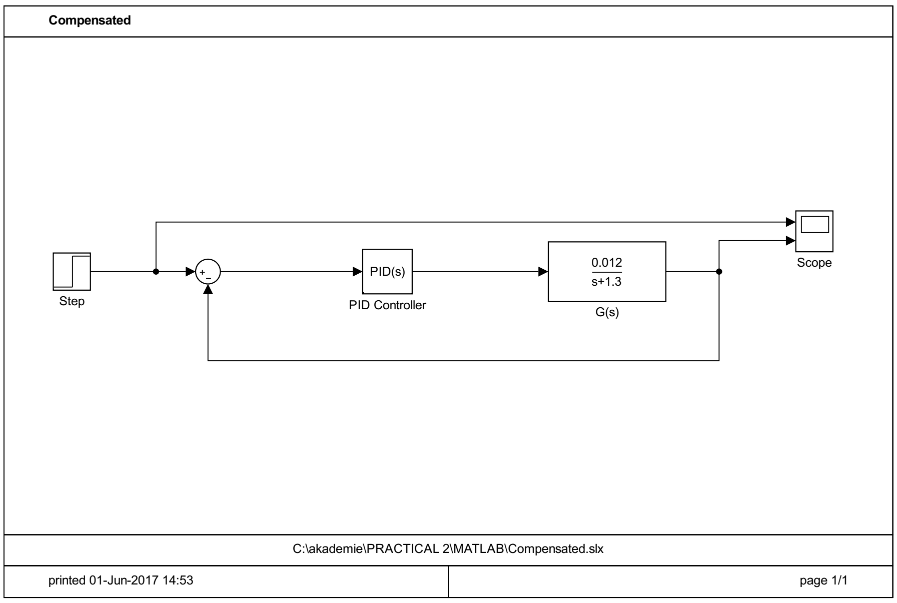
Table 2: Theoretical Results

Parameter	**Value**
Steady state error	0%
Percent Overshoot(P.O.)	10.5 %
Settling time	2.2s

CONTROLLER SIMULATION

Simulink automates most of the simulation process. Figure 4 Shows the simple simulation. The simulation was also used to tune the Parameters calculated in the previous section to be well within the specifications. These tuned parameters were implemented using the controller.The parameters after the tuning process are given by:

K_p	300
K_i	800



Compensated
Step
PID(s)
PID Controller
0.012
s+1.3
G(s)
Scope
C:\akademie\PRACTICAL 2\MATLAB\Compensated.slx
printed 01-Jun-2017 14:53
page 1/1

Figure 4: Simulink Block diagram of the compensated plant simulation

The simulation in Figure 4 yields the transient response in Figure 5

Table 3 Compares the specification to the simulated result of the calculated parameters:

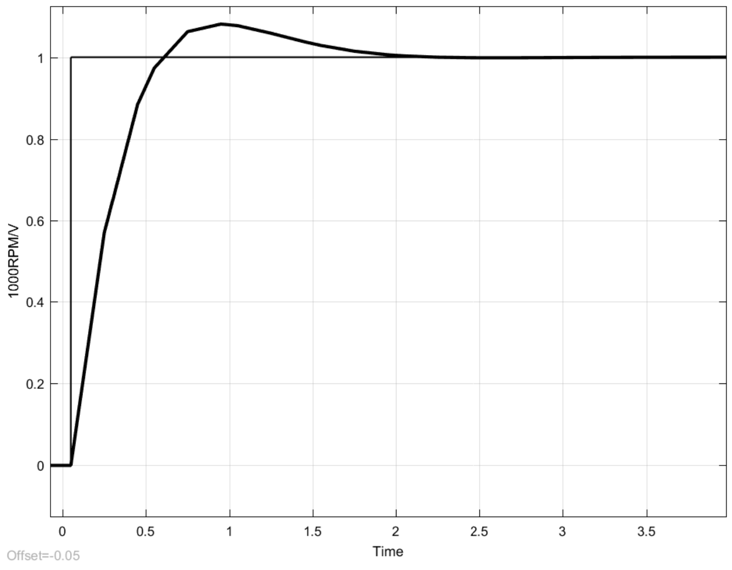


Figure 5: Plot of the transient response of the simulated system

Table 3: Simulated Results

Parameter	**Value**
Steady state error	0%
Percent Overshoot(P.O.)	8.1 %
Settling time	2s

Having calculated and simulated the PI compensator, it was implemented on an ARM microcontroller and connected to a driver circuit. The microcontroller was also configured to log the following data:

- The desired set point (Volt)
- Tachometer output (Volt)
- Error (Volt)
- Duty cycle(%)

The output used to plot Figure 6 was prepared for plotting using the python script included in Appendix B. A step input of 1V was used, making the setpoint for the controller to track 1000RPM. Table 4 shows the practical results.

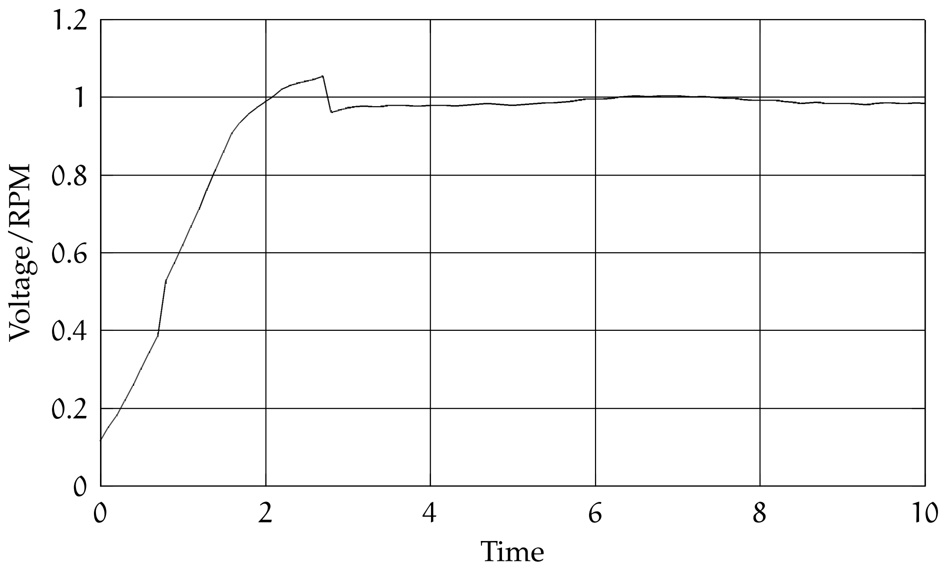


Figure 6: Practical measurement, transient step response

Table 4: Simulated Results

Parameter	**Value**
Steady state error	3%
Percent Overshoot(P.O.)	7.1 %
Settling time	6s

RESULTS

Table 5: Compiled Results

	P.O	E_{ss}	T_s
Specification	10%	1%	<2s
Calculation Results	10.5%	0%	2.2s
Simulation Results	8.1%	0%	2s
Practical Results	7.1%	3%	2.5s

DISCUSSION OF RESULTS AND CONCLUSION

The calculations, simulation and practical measurements all performed within a reasonable margin from the specifications. The steady state error in the practical measurements is due to inaccuracies in the measuring equipment and other environmental factors. The assignment was very educational and provided insights to a full stack controller implementation.

BIBLIOGRAPHY

[1] R. C. Dorf and R. H. Bishop, *Modern control systems*. Pearson, 2011.

[2] X. Li, "Simulink-based simulation of quadrature amplitude modulation (qam) system," *Proceedings of IAJC-IJME*, 2008.

APPENDIX A: ONE PAGE EXPERIMENT DESIGN

Table 6: Values to be measured/calculated

Variables and parameters	Symbol and unit
Armature Volatage	$v_a(t)[V]$
Armature Current	$i_a(t)[A]$
Motor Speed	$\omega(t) = \dot{\theta}(t)[rad/s]$
Armature Resistance	$R_a\ [\Omega]$

The purpose of this lab session is to design experiments to determine the time constants $\tau_a$and τ_l, where:

$$\tau_a = \frac{L_a}{R_a}$$
$$\tau_l = \frac{R_a J}{R_a b + K_b K_m}$$

This will be done using two experiments:

- First, the motor's speed will be measured as it slows down. This makes it possible to determine the moment of innertia and the mechanical time constant
- Second, the Speed will be realted to different values for the armature current and voltage.

These values will be digitally captured and used to determine the nessecary parameters.

APPENDIX B: SOURCE CODE LISTINGS

10.1 PYTHON AVERAGE FILTER

```python
import numpy as np
import csv

reader=csv.reader(open('Motor_start_all.tsv' ,"rb"),delimiter='	')
x = list(reader)
data = np.array(x)
rows = data.shape[0]
arr_time = data[:-(26) ,0].astype(float)
arr_volt = data[:,1].astype(float)
arr_rpm = data[:,2].astype(float)

def smooth(y, box_pts):
    box = np.ones(box_pts)/box_pts
    y_smooth = np.convolve(y, box, mode='same')
    return y_smooth

arr_volt_smooth = smooth(arr_volt,1000)
arr_rpm_smooth = smooth(arr_rpm,1000)




f1 = open('Motor_start_combined_smoothed.tsv',"wb")
writer = csv.writer(f1,delimiter = "	")
for i in range(0,len(arr_time)):
        writer.writerow((arr_time[i]-26,arr_volt_smooth[i],arr_rpm_smooth
            [i]))
```

10.2 STM32F SOURCE CODE (PID FUNCTION)

```c

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
        double dt = 100000*(0.000001);
        double sum =0;
                int k = 0;
        HAL_ADC_Start(&hadc1);
        HAL_ADC_Start(&hadc2);
        int duty = 50;
double div = 256/2.8667;
        double V1 = 0;
        double V2 = 0;
        char temp1[50];
        char temp2[50];
        V1 = (double)HAL_ADC_GetValue(&hadc1); //ADC on PA2
        V2 = (double)HAL_ADC_GetValue(&hadc2); //ADC on PA3

        while(k <10000)
        {
        sum += (double)HAL_ADC_GetValue(&hadc2); //ADC on PA3
        k++;
        }
        V2 = sum/10000;
        V1 = (V1/div);
        V2 = (V2/div);
        double ERROR = V1-V2;
        integral = integral + (ERROR*dt);
        double kp = 160;
        double ki = 90;
        double out = kp*ERROR + ki*integral;
        if(out-prePWM > 2){out = prePWM + 2;}
        if(out-prePWM < 2){out = prePWM - 2;}
        sprintf(temp1,"OUT = %f \t PREOUT = %f \t ERROR = %f \t TACHO = %f \t SOURCE = %f \n",(out),(prePWM),(ERROR),(V2), V1);
                if (out > 100){out = 100;}
        if (out < 0){   out = 0;}
        TIM1->CCR1 = round(100-out);
        prePWM = out;
//sprintf(temp1,"V1 = %f \n V2 = %f \n ERROR = %f \n INTEGRAL = %f \n OUT = %f \n",(V1),(V2),(ERROR),(integral),(out));
trace(temp1,50);
//MX_TIM1_Init();
        //MX_ADC1_Init();
        //trace("Interrupt!\n",11);
}
/* USER CODE END 4 */
```