VIETNAMESE-GERMAN UNIVERSITY

ECE2021

FINAL REPORT of DRIVES in AUTOMATION Control Speed of DC Motor

BY

Lê Tuấn Minh – 10221031

Lê Quốc Hoàng – 10221015

Lương Gia Bảo – 10221006

Trần Vũ Lân – 10221028

Trần Thế Dương - 10221054

Dr. Bùi Minh Dương

Bình Dương, (22/07/2024)

Contents

Chapter 1

Introduction

A fan is a machine with rotating blades that creates an airflow current. It is usually used for ventilation and cooling. The function of DC motors in fans is to convert electrical energy into rotational mechanical energy. Brushed DC motors are widely used in this situation because they have precise and smooth speed control. Customers can easily set the fan to the desired level of airflow, enhancing comfort and energy savings. On the other hand, DC motors have a small power-over-weight ratio, making them desirable for small load consumption and compact size design.

This experiment investigates the characteristics of separately excited DC motors and the application of DC motors as fan machines. We expected the 10% error due to the variety of 775 motors and inconsistent quality control.

Figure 1.1: Motor 775

Chapter 2

Theorotical Background

2.1 Operating Principle of Motor

In this implementation, the 775 DC motor, which is a brushed permanent magnet DC motor, is used. The stationary field system (stator) is created by a permanent magnet, while the armature winding forms a closed loop through the commutator (rotor). Brushes, which are positioned along the neutral axis on the commutator, supply power to the armature. The magnetic field produced by the stator (F_f) aligns with the magnetic axis, while the current flowing through the rotor generates another magnetic field (F_a) directed along the brush axis. These two magnetic fields are in spatial quadrature and act simultaneously within the motor. Their interaction results in torque, causing the armature to rotate. Additionally, an induced voltage—known as back EMF—occurs in the armature.

The 775 DC motor is classified as a separately excited DC motor, where the field winding circuit is separated from the armature.

Then, the power value is obtained by using the formula:

$$P = V \cdot I \tag{2.1}$$

The torque value is obtained by using the formula:

$$T = \frac{P}{\omega} = U \cdot \frac{I}{\omega} \tag{2.2}$$

The measurement values for Equation 3.1 and Equation 3.2 are represented in Figure 4.7.

The constant k of the DC motor is obtained by using the formula:

$$K = \frac{T}{I} \tag{2.3}$$

For DC motors to be implemented to work as a fan. DC motors should be capable of controlling speed with a constant torque through a control PWM pulse which changes the voltage applied to the motor. Two cases were examined here, the no-load case and the full-load case.

As previously mentioned, because flux ϕ is relatively constant, we can write the relationship between torque and current as $T = K_t \cdot I$ in which K_t is constant.

The formula to calculate speed is:

$$\omega = \frac{V}{K} \tag{2.4}$$

Firstly, for the no-load case, because torque T=0, hence $\omega = V/K$ or speed is directly proportional to voltage. Therefore, when a speed value is desired, the applied voltage is calculable:

$$V = \omega \cdot K \tag{2.5}$$

Secondly, for the full load case, the applied voltage is calculated:

$$V = \omega \cdot K + \frac{TR}{K} \tag{2.6}$$

Statistical analyses were performed using Excel.

2.2 Characteristic of a Separately Excited DC Motor

DC motor speed-torque and current torque characteristics are examined and the change of characteristic line when the applied voltage is changed.

2.2.1 Torque-Armature Current Characteristic

In the case of DC Separately excited motors, field flux ϕ is constant (neglecting armature reaction). Therefore, torque is proportional to armature current. Hence, the T-I_a characteristic for a DC separately excited motor will be a straight line through the origin.

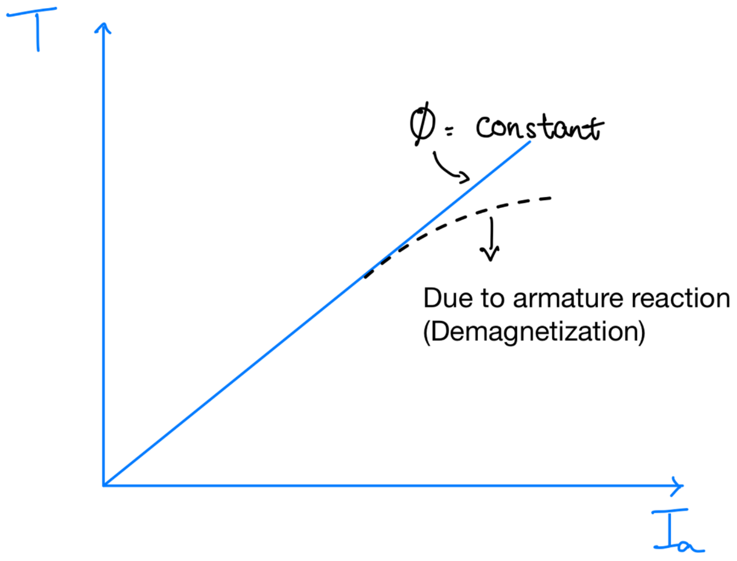


Figure 2.1: The T-I_a characteristic curve

2.2.2 Torque-Speed Characteristic

- A speed drop of 2 to 3% as the torque varies from no-load to full load.
- The armature MMF reacts with the field MMF which is known as the armature reaction. When the effects of the armature reaction are neglected, the flux per pole of the motor is constant and is independent of load.
- They move along the Y-axis (speed axis) following changes in the armature voltage.
- The field winding of the motor is supplied from a separate source. The smooth variation of armature voltage brings about the speed control in the zero to base speed range very efficiently.
- The motor operates in a constant torque mode.
- We assume that ϕ is constant in operation. In DC motor, $K_t = K_e$. Hence we have the following function:

$$N = \frac{V_a}{K_e\phi} - \frac{T_d}{K_t K_e \phi^2} r_a \tag{2.7}$$

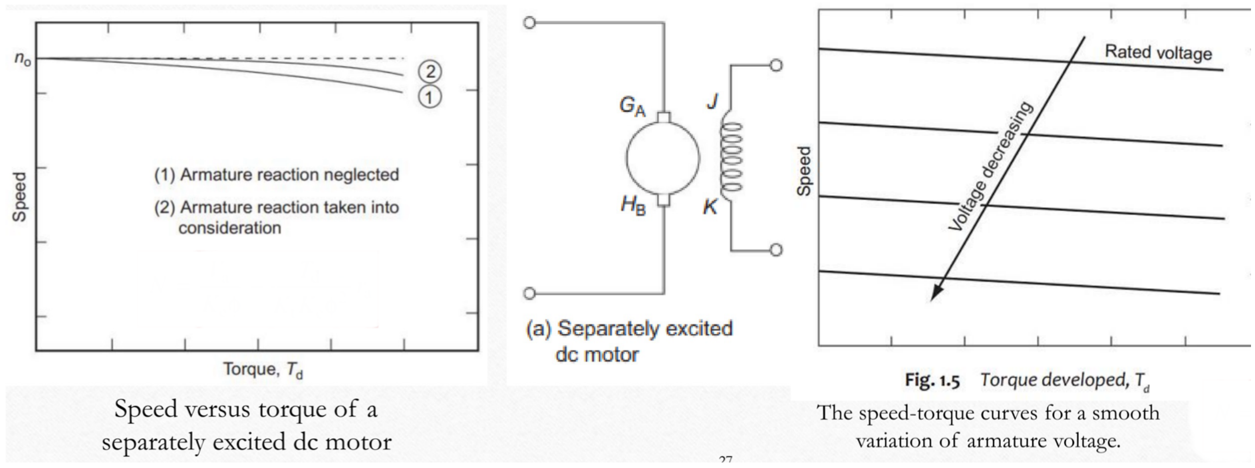

n_o
②
①
Speed
(1) Armature reaction neglected
(2) Armature reaction taken into consideration
Torque, T_d
Speed versus torque of a separately excited dc motor
G_A
H_B
J
K
(a) Separately excited dc motor
Rated voltage
Voltage decreasing
Speed
Fig. 1.5 Torque developed, T_d
The speed-torque curves for a smooth variation of armature voltage.

Figure 2.2: The Torque-Speed characteristic curve

2.3 Driver BTS7960

We use PWM modulation on the H bridge circuit to control the applied voltage to the DC motor. This is done by using the BTS7960 module.

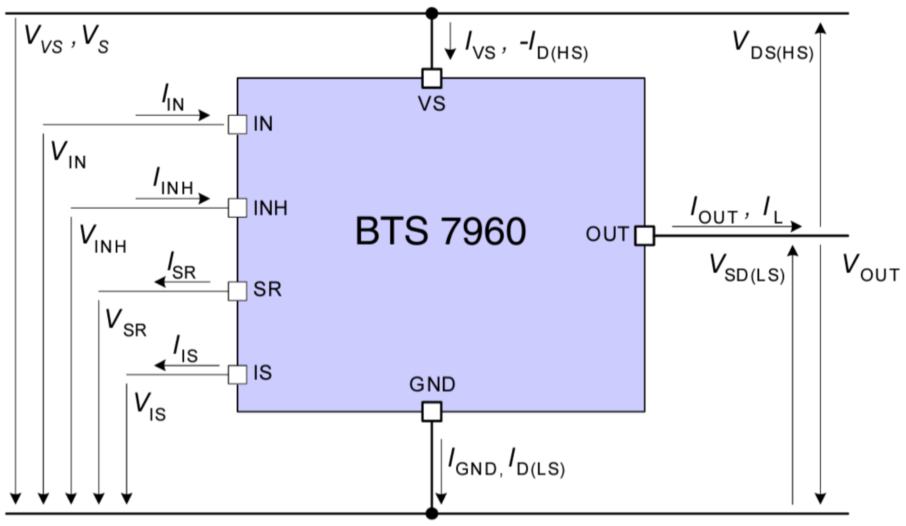


Figure 2.3: The diagram of BTS7960B

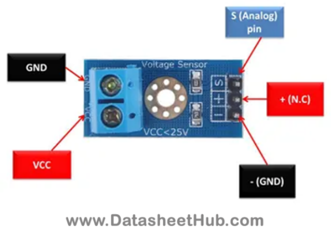
where:

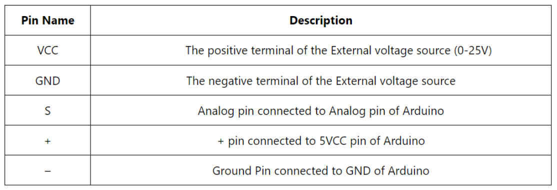
- VS is voltage input, in this case 24V
- OUT is the voltage output to the motor, which is regulated by PWM signal from Arduino using PWM modulation technique.
- V_S is the power supply of the module.
- INH, which is REN, LEN pins on the surface of the module, the BTS790 only works when INH pins are set HIGH
- IN is the PWM pin, which defines whether high- or low-side MOSFET is activated, which will be connected to digital output pins D9, and D10 of Arduino.

2.4 Sensors

To detect the voltage, the voltage sensor module DC 0-25V TH068 is used. Fundamentally, TH068 is a 5:1 voltage divider using a $30k\Omega$ and a $7.5k\Omega$ resistor. TH068 measured input voltage can be made 5 times smaller, hence with the limit of Arduino analogue input of 5 VDC only, the measured input voltage of TH078 can be 0-25V.

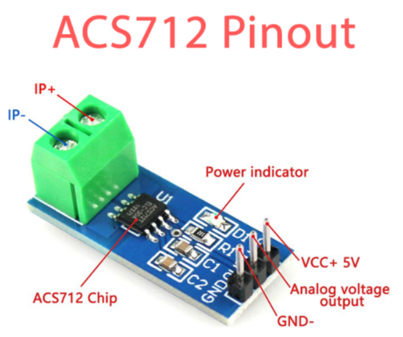
To detect the output current to the motor, the Hall effect current sensor ACS712 is used. The device consists of a linear Hall sensor circuit with a copper conduction path. Applied current flowing through this copper conduction path generates a magnetic field which is sensed by the integrated Hall IC and converted into a proportional voltage to the output pin, which will be connected to an arduino analog pin for measurement

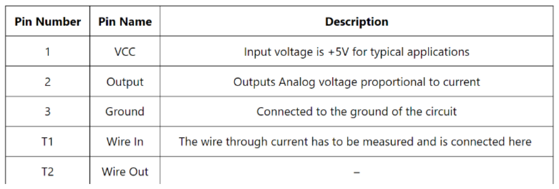




Pin Name	Description
VCC	The positive terminal of the External voltage source (0-25V)
GND	The negative terminal of the External voltage source
S	Analog pin connected to Analog pin of Arduino
+	+ pin connected to 5VCC pin of Arduino
–	Ground Pin connected to GND of Arduino

(a) Voltage Sensor





Pin Number	Pin Name	Description
1	VCC	Input voltage is +5V for typical applications
2	Output	Outputs Analog voltage proportional to current
3	Ground	Connected to the ground of the circuit
T1	Wire In	The wire through current has to be measured and is connected here
T2	Wire Out	–

(b) Current Sensor

Figure 2.4: Sensors

2.5 Tachometer

Figure 2.5: Tachometer

Tachometer also known as Rpm meter, digital tachometer. This device measures the rotation speed for all types of engines, the engine's shaft, or the rotation speed of the ball. Speed recorders are designed to be handheld to measure many types of engines and machinery.

Currently, rotation speed is determined by RPM (revolutions per minute) which is understood in terms of revolutions per minute. The measurement unit of RPM is calculated in (rpm). RPM is considered a common unit to calculate circular motion or the speed of movement of an object within 1 minute.

$$1\text{RPM} = 16.67mHz$$

As the RPM index gets larger as the engine is working more and more, the faster the engine operates. The engine's rotation speed indicates the engine's operating ability. In particular, checking the rotation speed will help users control the operating status of the device.

Chapter 3

Design and Implementation

3.1 Methodology

The experiment was performed indoors in room 414 in the dormitory of the Vietnamese German University of Ben Cat. Binh Duong Province, Viet Nam on the following days 23,25,27 July 2024. The temperature was between 29 to 34 degrees Celsius. The control tests for sensors were interspersed between each experimental test to compensate for the offset error of the sensors and the temperature variable would be approximately the same between experiment

Firstly, connect the components following this diagram.

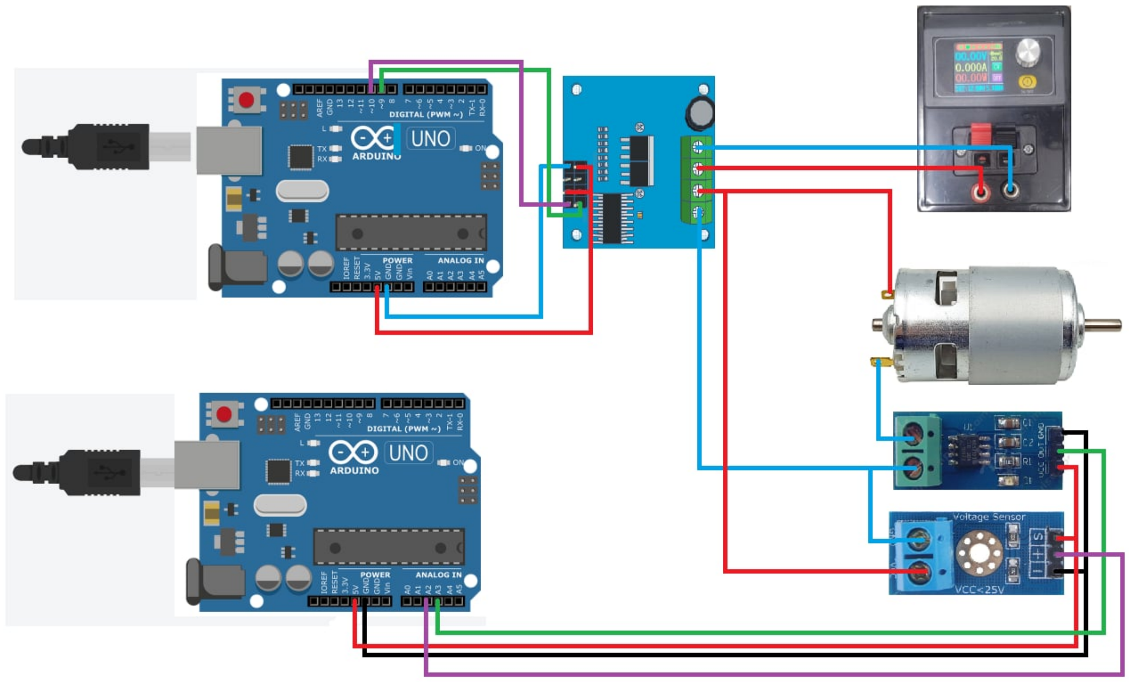


Figure 3.1: Configuration

3.2 Implementation

3.2.1 Arduino Code for Motor Controller

The Arduino code controls a DC motor using PWM (Pulse Width Modulation) signals and responds to commands received via serial communication (UART). It allows controlling the motor speed and direction remotely.

1. **Pin Definitions and Variables**

```
// PWM output pin
int pwm_pin_1 = 9;
int pwm_pin_2 = 10;

// LED pin
int led_pin = LED_BUILTIN;

// Motor state and speed variables
bool motorRunning = false;
bool motorForward = false;
int motorSpeed = 0;

```

- **PWM Pins**: pwm_pin_1 (Pin 9) and pwm_pin_2 (Pin 10) are used for PWM output to control motor speed.
- **LED Pin**: led_pin is the built-in LED pin, used for indicating motor activity.
- **Motor State Variables**: motorRunning, motorForward, and motorSpeed manage the motor's operational state.

2. **Setup Function (setup())**

```
void setup() {
    pinMode(pwm_pin_1, OUTPUT);
    pinMode(pwm_pin_2, OUTPUT);
    pinMode(led_pin, OUTPUT);
    TCCR2B = TCCR2B & B11111000 | B00000001; // prescaler of 1

    // begin serial communication
    Serial.begin(9600);
}

```

- **pinMode()**: Configures pins as outputs for PWM and LED.
- **Timer Configuration (TCCR2B)**: Sets up for PWM operation with a prescaler of 1.
- **Serial Communication (Serial.begin())**: Initializes serial communication with a baud rate of 9600.

3. **Main Loop Function (loop())**

```
void loop() {
    while (Serial.available() == 0) {}

    char incoming = Serial.read();

    if (incoming >= '0' && incoming <= '9') {
            motorSpeed = map(incoming, '0', '9', 0, 255);
        } else if (incoming == 'F') {
            motorRunning = true;
            motorForward = true;
        } else if (incoming == 'B') {
            motorRunning = true;
            motorForward = false;
        } else if (incoming == 'S') {
            motorRunning = false;
    }

    if (motorRunning) {
        if (motorForward) {
            analogWrite(pwm_pin_1, motorSpeed);
            analogWrite(pwm_pin_2, 0);
            digitalWrite(led_pin, HIGH);
        } else {
            analogWrite(pwm_pin_1, 0);
            analogWrite(pwm_pin_2, motorSpeed);
            blinkLED();
        }
    }

    else {
        analogWrite(pwm_pin_1, 0);
        analogWrite(pwm_pin_2, 0);
        digitalWrite(led_pin, LOW);
    }
}

```

- **Serial Input Handling**: Waits for serial data and reads it into **incoming**.
- **Command Interpretation**: Based on the received command (**incoming**):
  - Adjusts **motorSpeed** if a valid speed command is received.
  - Sets **motorRunning** and **motorForward** based on directional commands ('F' for forward, 'B' for backward, 'S' for stop).
- **Motor Control**: Controls PWM outputs (**analogWrite()**) to adjust motor speed and direction. LED Feedback: Lights up the LED (**digitalWrite()**) to indicate motor activity.

4. **blinkLED()** Function

```
void blinkLED() {
    static unsigned long lastBlink = 0;
    unsigned long currentMillis = millis();
    if (currentMillis - lastBlink >= 100) {
        lastBlink = currentMillis;
        digitalWrite(led_pin, !digitalRead(led_pin));
    }
}

```

- **blinkLED()**: A helper function that toggles the built-in LED (led_pin) every 100 milliseconds when the motor is moving backward (**motorForward** is false).

3.2.2 Arduino Code for Sensors

1. **Pin Definitions and Variables**

```
// Current sensor pin
int OutPin_Current = A3;

// Voltage sensor pin
const int voltageSensorPin = A2;

// Arduino constants
const float vCC = 5.00;
const float cur_offset = 0.52;
const float vol_multiplier = 5.08;

int counter = 0;
float test_vol = 0.0;
float test_cur = 0.0;

```

- **Sensor Pins**: OutPin_Current (A3) is connected to a current sensor, and voltageSensorPin (A2) reads the output of a voltage sensor.
- **Constants**: VCC represents the Arduino's operating voltage, cur_offset adjusts the current sensor output, and vol_multiplier scales the voltage sensor reading.

2. **Setup Function (setup())**

```
void setup() {
    TCCR2B = TCCR2B & B11111000 | B00000001; // prescaler of 1
    // begin serial communication
    Serial.begin(9600);
}

```

- **Timer Configuration (TCCR2B)**: Sets up Timer 2 with a prescaler of 1 for PWM operation.
- **Serial Communication (Serial.begin())**: Starts serial communication with a baud rate of 9600.

3. **Main Loop Function (loop())**

```
void loop() {
    float total_cur = 0.0;
    float total_vol = 0.0;

    // Average current over 1000 samples
    for (int x = 0; x < 1000; x++) {
    total_cur += (0.049 * analogRead(OutPin_Current) - 25);
    }

    // Average voltage over 1000 samples
    for (int x = 0; x < 1000; x++) {
    total_vol += analogRead(voltageSensorPin);
    }

    // Accumulate averaged current and voltage values
    test_cur += -(total_cur / 1000.0 + cur_offset);
    test_vol += (total_vol / 1000.0 / 1023.0) * vCC;

    counter++;

    // Print averaged current and voltage values every 10 cycles
    if (counter == 10) {
        Serial.print(test_cur / counter, 7);
        Serial.print(",");
        Serial.print(test_vol * vol_multiplier / counter, 7);
        Serial.print("\n");

        // Reset accumulators and counter
        test_vol = 0;
        test_cur = 0;
        counter = 0;
    }

    delay(1);  // Delay between loops for stability
}

```

- **Current Sensing**: Reads analog values from OutPin_Current, averages over 1000 samples, adjusts for offset (cur_offset).
- **Voltage Sensing**: Reads analog values from voltageSensorPin, averages over 1000 samples, calculates actual voltage using vol_multiplier.
- **Serial Output**: Outputs averaged current (test_cur) and voltage (test_vol) every 10 cycles (counter == 10) over serial communication.
- The formulas for these 2 sensors are obtained from the reference resources.

3.2.3 Python Code for Motor Controller and Real-time Plotter

The provided code is a Python program that integrates a Tkinter-based motor controller with a Matplotlib real-time plotter. The motor controller communicates with a DC motor through a serial connection on COM5, while the real-time plotter reads voltage and current data from an Arduino connected on COM6. Below is a detailed explanation of how the code functions.

1. **Importing Libraries**
   The code begins by importing the necessary libraries:

```
import serial
import tkinter as tk
from tkinter import ttk
import threading
import time
import matplotlib.pyplot as plt
import matplotlib.animation as animation

```

   - **serial**: For serial communication with the Arduino and motor controller.
   - **ttk**: For creating the graphical user interface (GUI).
   - **threading**: For running the plotter and GUI concurrently.
   - **time**: For adding delays, particularly for serial initialization.
   - **matplotlib.pyplot, matplotlib.animation**: For creating and updating the real-time plot.

2. **Tkinter GUI for Motor Control**
   The GUI is created using Tkinter, with a dark mode theme applied using ttk.Style.

   (a) **Creating the Main Window:**

```
root = tk.Tk()
root.title("Motor Controller")
root.configure(bg="#333")

```

   - ‘tk.Tk()‘ is a constructor that initializes the Tkinter framework and creates the root window, which serves as the main container for all the widgets (buttons, labels, etc.) in the application.
   - The title is set to "Motor Controller”
   - The bg parameter sets the background color of the window. Here, the background color is set to #333, which is a dark gray color.

   (b) **Setting Up Styles**:

```
style = ttk.Style()
style.theme_use("clam")
style.configure("TButton", background="#444", foreground="#fff")
style.configure("TScale", background="#444", foreground="#fff")

```

   - The style’s theme is set to "clam”, one of the available themes in ttk and is known for its clean and modern look.

- background="#444" sets the background color of the buttons and the scale to a dark gray.
- foreground="#fff" sets the text color of the buttons and the scale to white.

(c) **Establishing the Serial Connection**:

```
// Create a serial connection
ser = None

def connect_port():
    global ser
    try:
        ser = serial.Serial(port_var.get(), 9600, timeout=1)
        status_label.config(text="Connected", foreground="green")
    except serial.SerialException as e:
        status_label.config(text=f"Error: {e}", foreground="red")

// Port selection
port_label = ttk.Label(root, text="Select COM Port:")
port_label.pack(fill="x")
port_var = tk.StringVar()
port_entry = ttk.Entry(root, width=10, textvariable=port_var)
port_entry.pack(fill="x")
port_var.set("COM3")  // default port
connect_button = ttk.Button(root, text="Connect", command=
    connect_port)
connect_button.pack(fill="x")
status_label = ttk.Label(root, text="Not connected", foreground="
    red")
status_label.pack(fill="x")

```

- ser: A global variable intended to store the serial connection object. Initially, it is set to None until a connection is established.
- connect_port: This function tries to establish a serial connection using the port specified by the user in the GUI.
- global ser: Declares that the function will modify the global ser variable.
- ser = serial.Serial(port_var.get(), 9600, timeout=1): Attempts to open a serial connection on the port specified in port_var with a baud rate of 9600 and a timeout of 1 second.
- If the connection is successful, the status message will change to "Connected" in green.
- status_label.config: Updates the status label to inform the user whether the connection was successful or if an error occurred.
- port_label: A label widget that prompts the user to select the COM port.
- port_var: A Tkinter StringVar that holds the value of the selected port.
- port_entry: An entry widget where the user can type the COM port number.
- port_var.set("COM3"): Sets the default value of the COM port entry to "COM3".

- connect_button: A button widget that, when clicked, calls the connect_port function to establish the serial connection.
- status_label: A label widget that displays the connection status to the user, initially set to "Not connected" with red text indicating no connection.

(d) **Creating the Motor Control Frame**
The frame contains controls for setting speed, moving forward, backward, and stopping the motor.

```
motor_frame = ttk.Frame(root, padding="10")
motor_frame.pack(fill="x")

```

- ttk.Frame(root, padding="10"): Creates a new frame widget inside the main window (root) with a padding of 10 pixels.
- motor_frame.pack(fill="x"): Adds the frame to the window, expanding it horizontally to fill the available width.

(e) **Speed Control**
This section of the code deals with setting up the speed control for the motor, including creating the necessary GUI elements for user input and defining the functionality to send the speed settings to the motor controller.

```
speed_label = ttk.Label(motor_frame, text="Speed:")
speed_label.pack(side="left")
speed_entry = ttk.Entry(motor_frame, width=5)
speed_entry.pack(side="left")

def set_speed():
    try:
        speed = int(speed_entry.get())
        if 0 <= speed <= 9:
            ser.write(bytes([speed]))
        else:
            status_label.config(text="Speed must be 0-9",
    foreground="red")
    except ValueError:
            status_label.config(text="Invalid speed input",
    foreground="red")

speed_button = ttk.Button(motor_frame, text="Set Speed", command=
    set_speed)
speed_button.pack(side="left")

```

- speed_label: A label widget inside the motor_frame that prompts the user to enter the desired speed for the motor.
- speed_entry: An entry widget where the user can input the speed value. It is placed next to the label for easy access.
- set_speed: This function reads the speed value entered by the user and sends it to the motor controller through the serial connection.

- speed = int(speed_entry.get()): Attempts to convert the user's input to an integer.
- if 0 <= speed <= 9: Checks if the speed value is within the valid range (0 to 9).
- ser.write(bytes([speed])): Sends the speed value as a single byte to the motor controller.
- status_label.config(text="Speed must be 0-9", foreground="red"): Updates the status label to inform the user if the speed value is out of the valid range.
- except ValueError: Catches any errors that occur if the user's input is not a valid integer and updates the status label accordingly.
- speed_button: A button widget that, when clicked, calls the set_speed function to send the entered speed value to the motor controller. It is placed next to the speed entry field and label for a cohesive user interface.

(f) **Motor Direction Controls**
Buttons are provided to move the motor forward and backward and stop it.

```
def move_forward():
    ser.write(b"F")

    forward_button = ttk.Button(motor_frame, text="Forward",
   command=move_forward)
    forward_button.pack(side="left")

def move_backward():
    ser.write(b"B")

    backward_button = ttk.Button(motor_frame, text="Backward",
   command=move_backward)
    backward_button.pack(side="left")

def stop_motor():
    ser.write(b"S")

    stop_button = ttk.Button(motor_frame, text="Stop", command=
   stop_motor)
    stop_button.pack(side="left")

```

- ttk.Button(motor_frame, text="Forward", command=move_forward): Creates a button labeled "Forward" that calls the move_forward function when clicked, inside the motor_frame. And the same logic applies to the Backward and Stop button.
- move_forward, move_backward, stop_motor: Functions to send commands to the motor to move forward, backward, or stop.
- ser.write(b"F"), ser.write(b"B"), ser.write(b"S"): Sends the respective command byte to the serial device.

3. **Matplotlib Real-time Plotter**
   The plotter reads data from an Arduino on COM6 and displays it in real-time using Matplotlib.

   (a) **Global Variables for Plotting**

```
precision = 5
voltage_list = []
current_list = []
time_list = []
avg_voltage = 0
avg_current = 0

```

   (b) **Animation Function**
   The animate function reads data from the serial port, updates the lists for voltage and current, and redraws the plot.

```
def animate(i, voltage_list, current_list, avg_voltage,
   avg_current, time_list, ser_plot):
    arduino_string = ser_plot.readline().decode().strip()
    try:
        current_str, voltage_str = arduino_string.split(',')
        voltage = float(voltage_str)
        current = float(current_str)
        voltage_list.append(voltage)
        current_list.append(current)
        time_list.append(len(time_list))
        avg_voltage = sum(voltage_list) / len(voltage_list)
        avg_current = sum(current_list) / len(current_list)
    except ValueError:
        pass
    ax1.clear()
    ax2.clear()
    ax1.plot(time_list, voltage_list, 'b-', label='Voltage (V)')
    ax2.plot(time_list, current_list, 'r-', label='Current (A)')
    ax1.set_ylim([4, 7])
    ax2.set_ylim([0, 2])
    ax1.set_xlabel("Time")
    color = 'tab:blue'
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.set_ylabel("Voltage (V)", color=color)
    ax1.set_title('Avg. Voltage: 'f'{avg_voltage:.{precision}f}''
    (V)', color=color, loc='left')
    color = 'tab:red'
    ax2.tick_params(axis='y', labelcolor=color)
    ax2.secondary_yaxis('right').set_ylabel("Current (A)", color=
   color)
    ax2.set_title('Avg. Current: 'f'{avg_current:.{precision}f}''
    (A)', color=color, loc='right')
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper left', bbox_to_anchor=(0.0, 0.9))

```

i. **Reading and Parsing Data**:
   - arduino_string = ser_plot.readline().decode().strip(): Reads a line of data from the serial connection (ser_plot), decodes it from bytes to a string, and removes leading/trailing whitespace.
   - current_str, voltage_str = arduino_string.split(','): Splits the string by comma to separate the current and voltage values.
   - voltage = float(voltage_str), current = float(current_str): Converts the parsed string values to floating-point numbers (floats).

ii. **Appending Data to Lists**:
   - voltage_list.append(voltage), current_list.append(current): Appends the newly acquired voltage and current values to their respective lists.
   - time_list.append(len(time_list)): Appends the current length of time_list, effectively adding a timestamp.

iii. **Calculating Averages**:
   - avg_voltage = sum(voltage_list) / len(voltage_list): Calculates the average voltage from all values in voltage_list.
   - avg_current = sum(current_list) / len(current_list): Calculates the average current from all values in current_list.

iv. **Plotting Updates**:
   - ax1.clear(), ax2.clear(): Clears the plots on both axes (ax1 and ax2) to prepare for new data.
   - ax1.plot(time_list, voltage_list, 'b-', label='Voltage (V)'): Plots the time vs. voltage data on ax1 with a blue line.
   - ax2.plot(time_list, current_list, 'r-', label='Current (A)'): Plots the time vs. current data on ax2 with a red line.
   - ax1.set_ylim([4, 7]), ax2.set_ylim([0, 2]): Sets the y-axis limits for both plots to predefined ranges.

v. **Styling and Labels**:
   - ax1.set_xlabel("Time"): Sets the x-axis label to "Time".
   - ax1.set_ylabel("Voltage (V)", color=color): Sets the y-axis label for ax1 to "Voltage (V)" with a specified color.
   - ax1.set_title('Avg. Voltage: 'f'avg_voltage:.precisionf" (V)', color=color, loc='left'): Sets the title for ax1 displaying the average voltage with specified precision and color.
   - Similar configurations are applied to ax2 for current.

vi. **Legends**:
   - ax1.legend(loc='upper left'), ax2.legend(loc='upper left', bbox_to_anchor=(0.0, 0.9)): Adds legends to ax1 and ax2, specifying their positions for the plots.

(c) **Start Plot Function**

```
def start_plot():
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    ser_plot = serial.Serial("COM6", 9600)
    time.sleep(2)
    ani = animation.FuncAnimation(fig, animate, fargs=(
   voltage_list, current_list, avg_voltage, avg_current,
   time_list, ser_plot), interval=100)
    plt.show()
    ser_plot.close()

```

The start_plot function initializes a Matplotlib figure with two y-axes, establishes a serial connection to COM6 for reading data from an Arduino, sets up a real-time animation to update voltage and current plots, displays the animated plot, and closes the serial connection when done.

(d) **Running the Plotter in a Separate Thread**:
The plotter is run in a separate thread to allow concurrent execution with the Tkinter main loop.

```
plot_thread = threading.Thread(target=start_plot)
plot_thread.start()

```

4. **Main Loop and Cleanup**
The Tkinter main loop is started, and the serial connection for the motor controller is closed when the GUI is closed.

```
root.mainloop()
ser_motor.close()

```

As seen in Figure 3.2 and Figure 3.3, this is our DC motor control and sensor control via Python workflow.

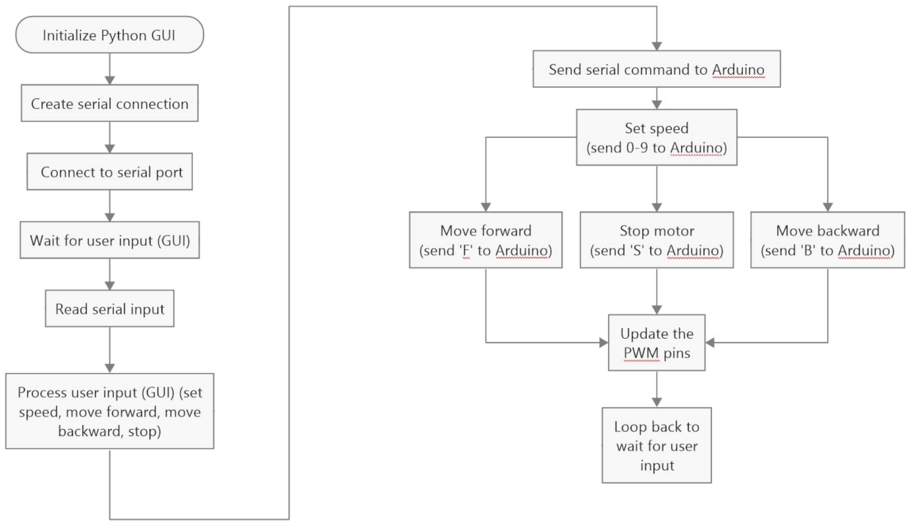

Initialize Python GUI
Create serial connection
Connect to serial port
Wait for user input (GUI)
Read serial input
Process user input (GUI) (set speed, move forward, move backward, stop)
Send serial command to Arduino
Set speed (send 0-9 to Arduino)
Move forward (send 'F' to Arduino)
Stop motor (send 'S' to Arduino)
Move backward (send 'B' to Arduino)
Update the PWM pins
Loop back to wait for user input

Figure 3.2: Motor Flowchart code

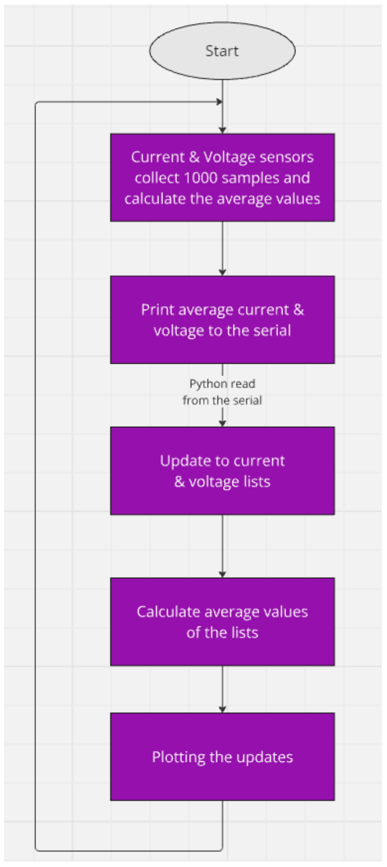

Start
Current & Voltage sensors collect 1000 samples and calculate the average values
Print average current & voltage to the serial
Python read from the serial
Update to current & voltage lists
Calculate average values of the lists
Plotting the updates

Figure 3.3: Sensor Flowchart code

3.3 Measurement

We want to measure the no-load operation at different voltages, and then measure the operations when load is increased for several levels at different voltages. For no load measure, apply nothing on the shaft of the motor, and begin measuring no-load voltage, no-load current, and no-load speed at different voltage levels (1,3,7,9). For load measure at different voltage levels, hang a water bottle on the shaft, to measure the operation point of the motor at different torque. The torque to the motor shaft is adjusted by pouring more water into the bottle, which makes the motor shaft heavier. This step is continued for different voltage levels (1,3,7,9) and speed, current, voltage data is obtained from the sensor.
Then, hang a water bottle on the shaft, to measure the operation point of the motor at different torque. The torque to the motor shaft is adjusted by pouring more water into the bottle, which makes the motor shaft heavier. This step is continued for different voltage levels (1,3,7,9) and speed, current, voltage data is obtained from the sensor.

In theory, we can calculate the theoretical voltage applied to DC motors as the PWM active time value is modulated.

$$V_{theory} = \text{PWM value} \cdot \frac{12}{10} \tag{3.1}$$

Voltage level	PWM duty cylce	Voltage by theory
1	10%	1.20
2	20%	2.40
3	30%	3.60
4	40%	4.80
5	50%	6.00
6	60%	7.20
7	70%	8.40
8	80%	9.60
9	90%	10.80

Table 3.1: Voltage value

For calculation, recalled equation (2.6) we can calculate average resistance which is relatively constant if we neglect the effect of heat during DC motor operation.

$$R = \frac{V - E}{I} \tag{3.2}$$

Recalled equation (2.4), we can find the DC motor constant:

$$K = \frac{\omega}{V} \tag{3.3}$$

In an attempt to investigate the speed control ability of the circuit. We separate the calculation into 2 situations, no-load and full-load operation. For a known speed desired, we calculate voltage and then find the measurement voltage that results in our known desired speed. Then we calculate the percentage error.

The calculated voltage is recalled from equation (2.6). For no-load operation.

$$V = \omega \cdot K \tag{3.4}$$

For full-load operation

$$V = \omega \cdot K + \frac{T.R}{K} \tag{3.5}$$

Percentage error is calculated by:

$$\frac{\text{Voltage Measured} - \text{Voltage Calculated}}{\text{Voltage Calculated}} \cdot 100 \tag{3.6}$$

Chapter 4

Results and Discussion

4.1 Table Values

Speed Desired	Voltage Calculated	Voltage Measured	Percent Error(%)
900.0000	1.3500	1.3300	1.5038
2940.0000	4.0200	4.0000	0.5000
6900.0000	10.3500	9.3300	10.9325
8600.0000	12.9000	12.0000	7.5000

Table 4.1: No-Load Value

Speed Desired	Voltage Calculated	Voltage Measured	Percent Error(%)
440.0000	1.2200	1.1917	2.3197
1940.0000	3.7700	3.4417	8.7082
5400.0000	8.8200	8.6317	2.1349
7390.0000	11.6100	11.6167	-0.0577

Table 4.2: Full-Load Value

Table 4.1 is the measurement and calculation of speed control attempts in the no-load situation.

Table 4.2 is the measurement and calculation of speed control attempts in a constant load situation.

Table 4.3 is the measurement values gained from the measurement procedure in part 3.3 Measurement of Chapter 3 Methodology.

Table 4.4 is the calculation from the values of table 4.3 where we calculate DC motor constant K, the EMF E values, the the armature resistance With the voltage, current, and speed

From measurement from Table 4.3, recalled equations (2.1) and (2.2) we can calculate the torque and power.

From Table 4.4, recalled equation (3.3), we can calculate the average DC motor constant.

$$K = 0.0015$$

Recall equation (3.2) we can calculate average resistance which is relatively constant if we neglect the effect of heat during DC motor operation.

$$R_{avg} = 0.2417$$

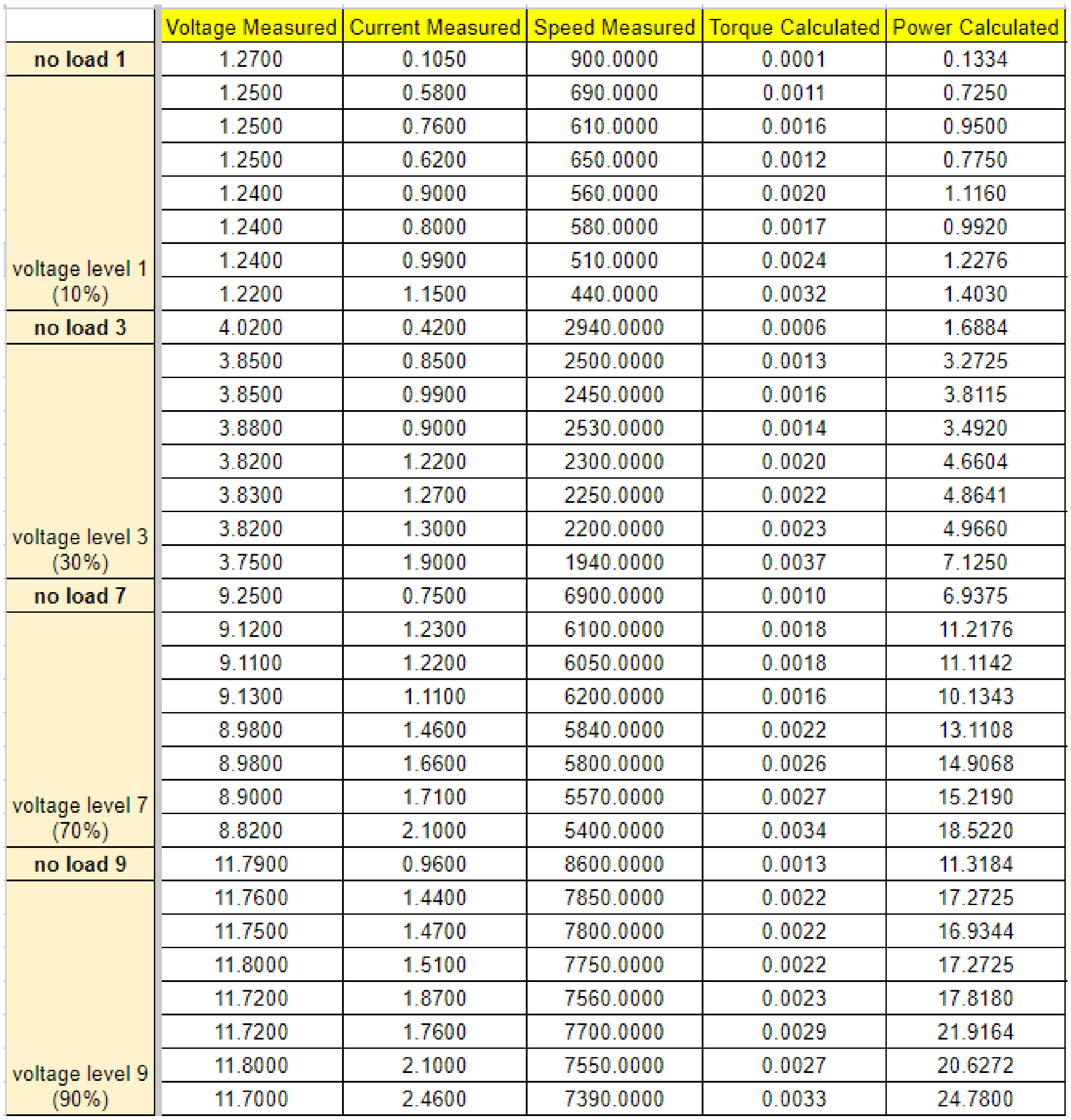
The average torque at full load is also calculated:

$$T_{avg} = 0.0033$$

Hence we can calculate this constant $\frac{T \cdot R}{K}$

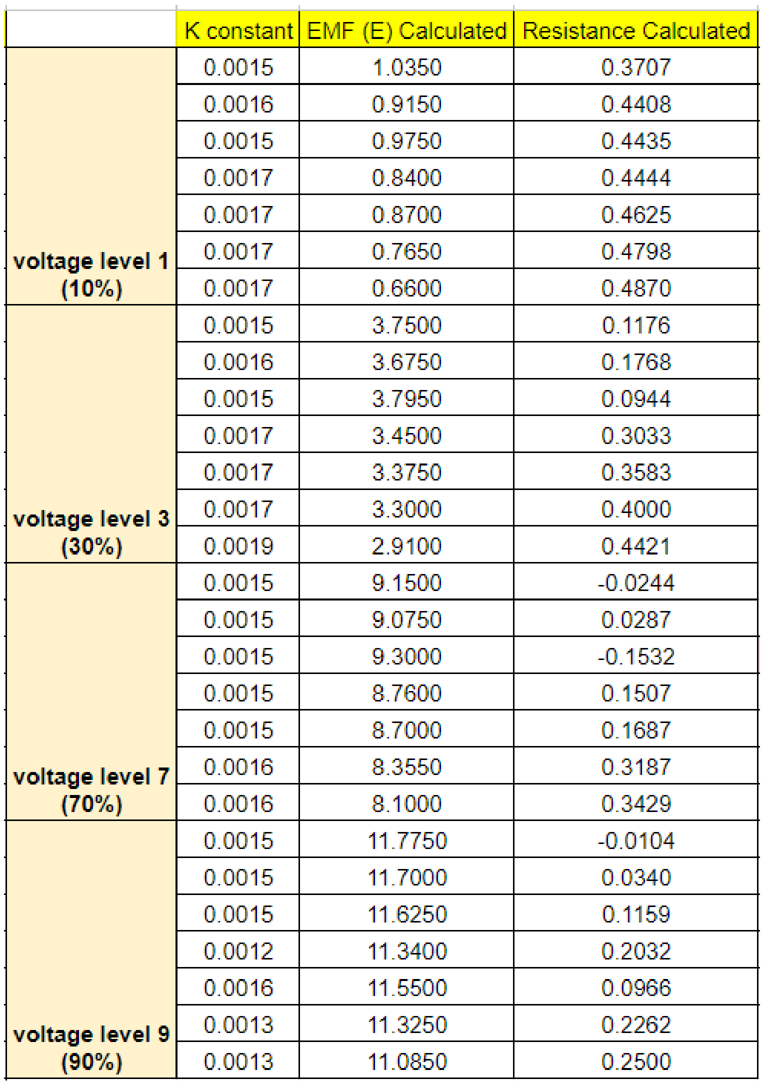
$$\frac{T \cdot R}{K} = 563.8560706$$

To investigate the speed control of the circuit, recalled equations (3.4), (3.5), and (3.6) and we gained Table 4.1 and Table 4.2



	Voltage Measured	Current Measured	Speed Measured	Torque Calculated	Power Calculated
no load 1	1.2700	0.1050	900.0000	0.0001	0.1334
	1.2500	0.5800	690.0000	0.0011	0.7250
	1.2500	0.7600	610.0000	0.0016	0.9500
	1.2500	0.6200	650.0000	0.0012	0.7750
	1.2400	0.9000	560.0000	0.0020	1.1160
	1.2400	0.8000	580.0000	0.0017	0.9920
	1.2400	0.9900	510.0000	0.0024	1.2276
voltage level 1 (10%)	1.2200	1.1500	440.0000	0.0032	1.4030
no load 3	4.0200	0.4200	2940.0000	0.0006	1.6884
	3.8500	0.8500	2500.0000	0.0013	3.2725
	3.8500	0.9900	2450.0000	0.0016	3.8115
	3.8800	0.9000	2530.0000	0.0014	3.4920
	3.8200	1.2200	2300.0000	0.0020	4.6604
	3.8300	1.2700	2250.0000	0.0022	4.8641
	3.8200	1.3000	2200.0000	0.0023	4.9660
voltage level 3 (30%)	3.7500	1.9000	1940.0000	0.0037	7.1250
no load 7	9.2500	0.7500	6900.0000	0.0010	6.9375
	9.1200	1.2300	6100.0000	0.0018	11.2176
	9.1100	1.2200	6050.0000	0.0018	11.1142
	9.1300	1.1100	6200.0000	0.0016	10.1343
	8.9800	1.4600	5840.0000	0.0022	13.1108
	8.9800	1.6600	5800.0000	0.0026	14.9068
	8.9000	1.7100	5570.0000	0.0027	15.2190
voltage level 7 (70%)	8.8200	2.1000	5400.0000	0.0034	18.5220
no load 9	11.7900	0.9600	8600.0000	0.0013	11.3184
	11.7600	1.4400	7850.0000	0.0022	17.2725
	11.7500	1.4700	7800.0000	0.0022	16.9344
	11.8000	1.5100	7750.0000	0.0022	17.2725
	11.7200	1.8700	7560.0000	0.0023	17.8180
	11.7200	1.7600	7700.0000	0.0029	21.9164
	11.8000	2.1000	7550.0000	0.0027	20.6272
voltage level 9 (90%)	11.7000	2.4600	7390.0000	0.0033	24.7800

Table 4.3: Measurement table value



	K constant	EMF (E) Calculated	Resistance Calculated
voltage level 1 (10%)	0.0015	1.0350	0.3707
	0.0016	0.9150	0.4408
	0.0015	0.9750	0.4435
	0.0017	0.8400	0.4444
	0.0017	0.8700	0.4625
	0.0017	0.7650	0.4798
	0.0017	0.6600	0.4870
voltage level 3 (30%)	0.0015	3.7500	0.1176
	0.0016	3.6750	0.1768
	0.0015	3.7950	0.0944
	0.0017	3.4500	0.3033
	0.0017	3.3750	0.3583
	0.0017	3.3000	0.4000
	0.0019	2.9100	0.4421
voltage level 7 (70%)	0.0015	9.1500	-0.0244
	0.0015	9.0750	0.0287
	0.0015	9.3000	-0.1532
	0.0015	8.7600	0.1507
	0.0015	8.7000	0.1687
	0.0016	8.3550	0.3187
	0.0016	8.1000	0.3429
voltage level 9 (90%)	0.0015	11.7750	-0.0104
	0.0015	11.7000	0.0340
	0.0015	11.6250	0.1159
	0.0012	11.3400	0.2032
	0.0016	11.5500	0.0966
	0.0013	11.3250	0.2262
	0.0013	11.0850	0.2500

Table 4.4: Calculation from Table 4.3

4.2 Characteristic Curve

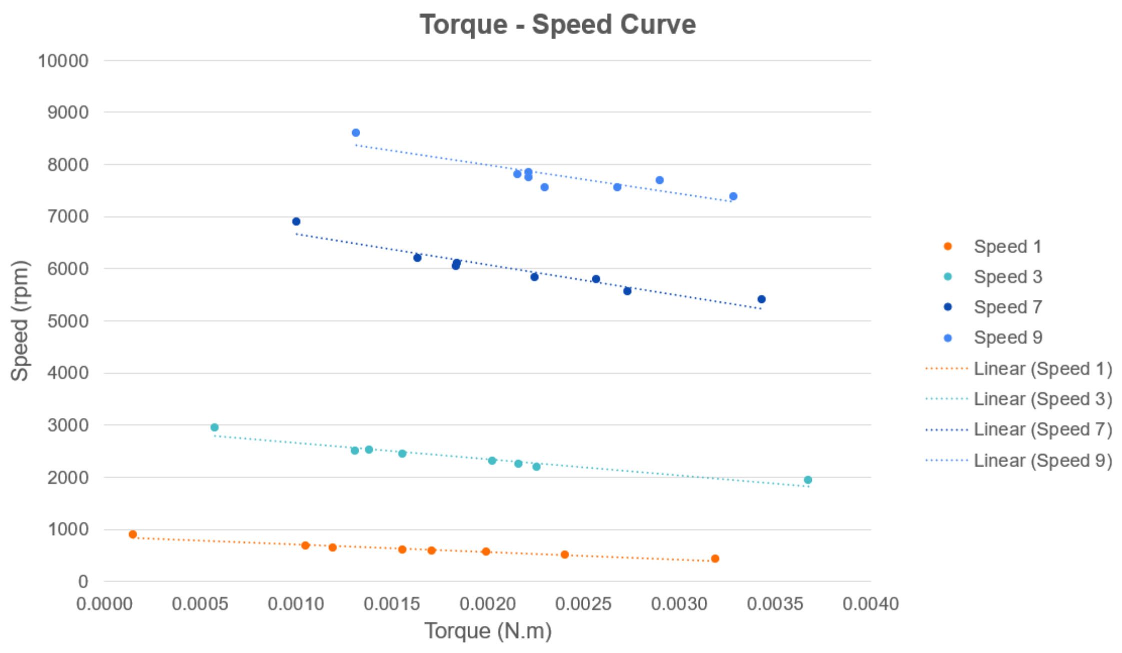


Figure 4.1: Torque Speed Curve

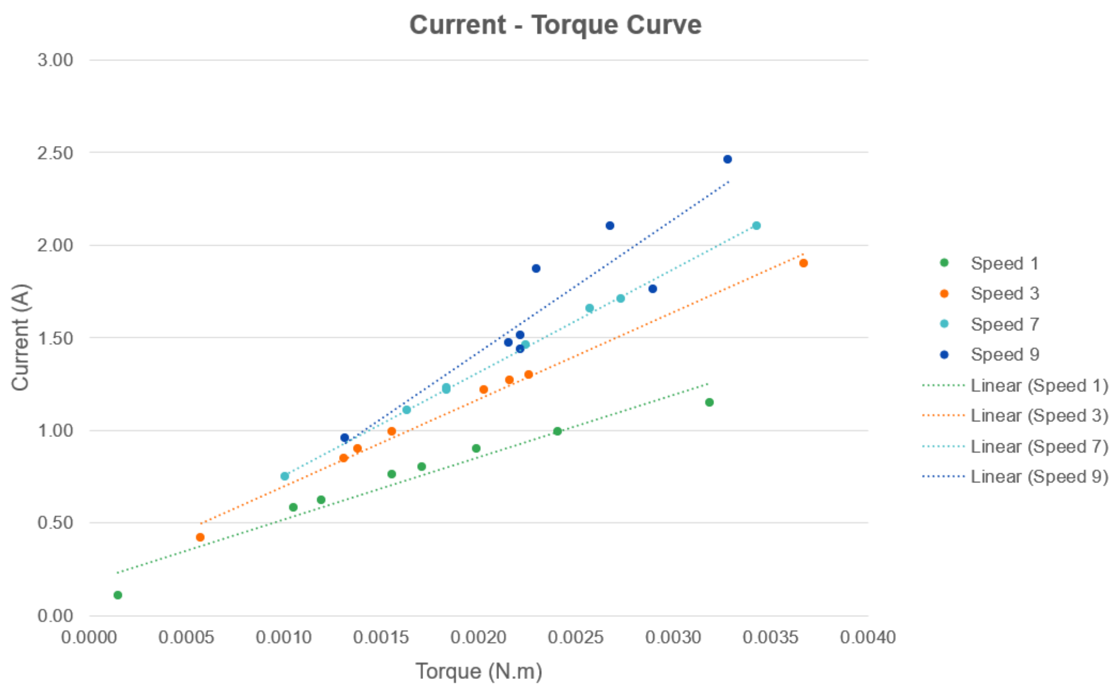


Figure 4.2: Current Torque Curve

4.3 Discussion

- The torque-speed curve is expected, and the rotational speed of the motor is inversely proportional to the torque value. The speed drops slightly as the torque increases from no load to full load. As the voltage level increases from 1 to 9, the no-load point gets higher which results in different characteristic lines.

- The I-T curve is expected, the torque is proportional to the armature current (I_a) in the absence of considering the armature reaction. The flux (ϕ) is assumed to remain constant, which leads to a linear relationship passing through the origin.

- There is inaccuracy in the measurement. One reason for this is heat due to the DC motor operating for a long time. Because of heat, which is easier to see at higher speeds, the resistance of the motor increases giving inaccuracy characteristic lines. Another reason is that the resistance of sensors and the H-Bridge module gives inaccurate readings. Another reason is that when DC motors operate at a very high speed, the torque produced by the motor shakes the motor itself which makes the speed measurement of the tachometer harder. One reason that must be taken into account is the inaccuracy of human oversight and other mistakes while reading and recording.

- The control of speed through changing the applied voltage by PWM modulation for the application of fan machines has small deviations of 10% or lower, except for the case of full load torque operation at a small speed.

- Further experiments with a larger range of values and precise procedures should be undertaken in order to explore the characteristics of DC motors and the application of fan machines by PWM modulation more efficiently.

Chapter 5

Conclusion

The characteristics of separately excited DC motors and the use of PWM modulation with H-Bridge module to control speed in fan machine application were examined. The error in using PWM modulation is 10% or lower except in the case of full load torque operation at a small speed, which is acceptable. The results were found to be affected by the generated heat of the DC motor at high speed and the inaccuracy of the sensor modules. The suggested approach to have a cooling system for the DC motor might further improve the accuracy.

References

1. https://image.dfrobot.com/image/data/DRI0018/BTS7960.pdf

2. https://www.datasheethub.com/acs712-5a-range-current-sensor-module/

3. https://www.youtube.com/watch?v=I9mDzI8at4I&ab_channel=Dr.ShyamSunderSharma