Evaluation of three phase motors

Categorisation of local magnetic polarity combinations

Josef Kartomi Thomas

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

EEBE UPC

Evaluation of three phase motors

Categorisation of local magnetic polarity combinations

by

Josef Kartomi Thomas

Student Name	Student Number
Josef Kartomi Thomas	590152

Instructor:	Ramon Bargallo
Lab technician:	Jordi Vilanova
Project Duration:	March, 2022 - June, 2022
Faculty:	Electrical engineering, EEBE UPC

Cover:	The Rolex Learning Center at EPFL (Modified)
Style:	EPFL Report Style, with modifications by Batuhan Faik Derinbay

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

Preface

Practical knowledge is becoming more and more important as I near my graduation. While I feel I have quite a broad understanding of electrical engineering, it's hard to concretely picture using this understanding in a professional setting. This project is therefore a supplement for my theoretical classes and a means for me to explore what it is to put into use practice knowledge in order to contribute to something of use. The project unfortunately started late due to some miscommunication but luckily professor Bargallo was nice enough to accommodate the rushed timeline.

This is a first attempt at creating something useful with my skills, and I hope that the project is able to reach the requirements and use sophisticated but simple techniques to achieve this. I appreciate your interest in reading this report and am open to criticism or questions which may be sent to josef.thomas@epfl.ch.

Josef Kartomi Thomas
EEBE, July 2022

Contents

List of Figures

1

Introduction

Currently electric motors already contribute a large part towards driving the world across many different domains. With every technological improvement, the world only steps towards a higher level of dependence on this technology. From electric cars to small consumer electronics, electric motors are becoming essential to humanity.

This project was therefore born in the Energy laboratory of EEBE due to the necessity to study and improve 3 phase motors and in particular in an efficient manner. Additionally, the project serves as a pedagogical exercise for a final year bachelor student in order to expand his practical skills underneath the supervision of an expert.

Objective

In order to improve motors quickly and efficiently before even testing the characteristics of a prototype, it's necessary to be certain that the prototype actually corresponds to desired design. While it's fairly obvious why this is the case, it is an important step in manufacturing and testing to know that the physical prototype matches the blueprints. This project is hence aimed at achieving this verification in the case of 3 phase motors. It will be further explained later in the report, but the project will study the use of local magnetic poles induced when current flows through the windings of the motor as a means to verify the motor.

The project is based on a machine made by a laboratory outside of EEBE and hence the majority of the work contained in this report will be concerning the software. In turn, the report will explore efficient means of evaluating a motor while maintaining robustness and above all methods that facilitate the ease of use of final machine.

***Note:** Italicised text in the software section often indicates a link to further information/explanation on the content in question.*

2

The working principle

2.1. Three phase motors

Without entering into too much detail, this type of motor works with 3 AC signals dephased by $pi/3$ rad. Generally, the signals are provided to different windings of the motor in a symmetrical and procedural design as seen in figure 2.1. These signals are split depending on the form of the inside of the motor but the overall principle is the same.

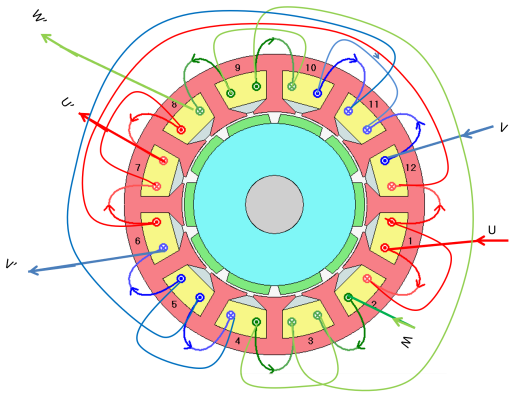


Figure 2.1: Correct stator winding

Each winding inside the stator produces a magnetic field according to Lenz's law [8]. Independent of the number of windings, the overall magnetic field can be calculated by taking the sum each individual magnetic field within each rotor winding at a given time. This overall magnetic field has both an amplitude (i.e. the strength of the field) and a direction (i.e. where the north and south poles are located). This property is exploited by the 3 phase current system because it is easy to create a rotating overall magnetic field while maintaining the same amplitude [6]. The function of this depends on the design of the rotor, but in essence the rotor follows a similar path the magnetic field of the rotor as it attempts to align itself. As the stator's magnetic field is rotating, the rotor will follow a similar path, and hence a rotational force is created.

2.2. The method

The stators explained above also produce magnetic fields when provided DC current and this principle is what will be exploited by the machine. With a fairly low amount of current, magnetic fields can be generated around each winding whose direction can be easily measured electrically (Hall sensors). Consequently, as 3 phase motors are manufactured in a specific form, it is possible to predict the shape of the induced magnetic field around the circular axis of the stator. One can visualise this using the green point in the image in 2.2 and following the line anti-clockwise while plotting the amplitude of the magnetic field at each point.

Therefore, the basic idea is to:

- Supply enough DC current to the motor in order to induce magnetic fields which can be measured by Hall sensors.
- Measure the magnetic field around the circular axis as explained above.
- Check which parts of the field do not coincide with the desired magnetic field.
- Knowing the configuration of the motor, deduce which coils are wound incorrectly.

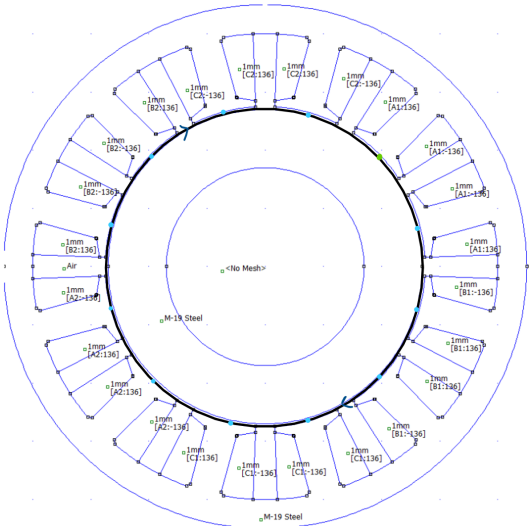


Figure 2.2: Stator measurement diagram

2.3. The specifics of the project

This project studies in particular 3 phase motors that have 12 stator windings. Like all 3 phase motors, this type has 3 lines dephased by $\frac{\pi}{3}$ rad which will be called A, B, and C that begin in the windings A1, B1, and C1 and end in the windings A2, B2, and C2 as seen in figure 2.1. However, in this case U, V and W represent A, B and C respectively.

Additionally, the points at which the magnetic field is maximal or minimal are found in specific positions which are very easy to measure. As each A1, B1, C1, A2, etc contains 2 coils, the total amount of coils are 12. Therefore using only 12 measurements of the magnetic field, the order and orientation of each winding can be determined. As the correct configuration of the windings is known, it is directly possible to deduce which lines are incorrectly wound.

A correctly wound motor will have a magnetic field analogous to the one in figure 2.3. By measuring the magnetic field at the points shown in blue in figure 2.2, it is possible to extract a binary representation where 1 signifies an overall local north pole and 0 a south one. This is useful later on when working with the micro-controller.

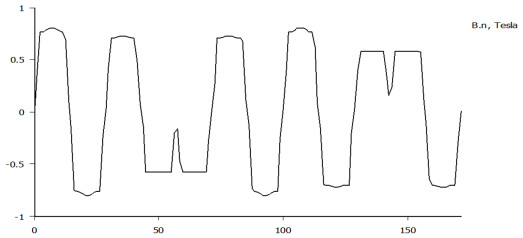


Figure 2.3: Magnetic field of correctly wound motor

In order to be able to categorise properly each improper configuration (i.e. one or multiple windings wound incorrectly) simulations were run by Professor Ramon Bargallo and the results of these simulations were then converted to digital values by checking the polarity at the blue points in figure 2.2 and then used to create table B.1. This table is directly used in the program discussed in a later section of the report. An example of an incorrectly wound motor can be found below in figure 2.4.

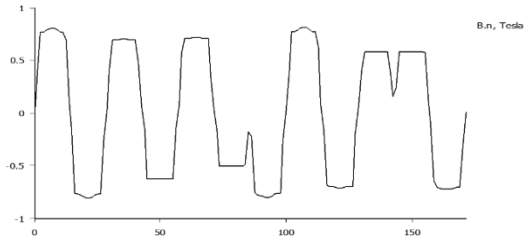


Figure 2.4: Magnetic field of motor where B1 and B2 are inverted

3

Hardware

As explained before, software was the focus of this project and little work was done on the hardware. There were however several unexpected problems that arose, which required solutions involving hardware modifications. A more in depth explanation of their function and feedback behaviours can be found in the README.md.

3.1. Sensors

3.1.1. Hall sensors (A3144)

This type of hall sensor [5] is digital and only senses whether the magnetic field is above or below a certain threshold. By placing one next to each winding, it is therefore possible to measure the local magnetic field and set its output pin to 1 or 0 depending on the magnetic field's overall direction. 12 of these are used to measure the local magnetic pole at each blue point in figure 2.2.

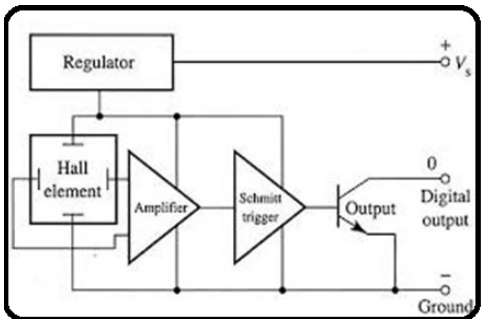


Figure 3.1: Digital hall sensor diagram
[5]

3.1.2. Piezo-electric buzzer

A simple audible feedback for each measurement forgoes the need to visually check the machine in order to know whether the motor is correct or not. This buzzer [2] is hence used to this means as it is not only simple to implement but cheap and power efficient. Separate and easily recognisable sounds are emitted for the case where the motor is correct and when it is not.

Figure 3.2: Passive buzzer
[2]

3.1.3. NeoPixel Ring

The NeoPixel Ring [7] used in the project consists of 12 LEDs connected in series, controlled through an i2c bus. The colour of each LED can be controlled individually with colours of a large variety. The purpose of these is to indicate which windings are correct visually. This avoids having to analyse the measurement as the evaluation is given next to each winding, allowing the user to pinpoint which one needs to be fixed. Correct local magnetic poles will have their corresponding LED turn green and incorrect ones will have theirs turn red.

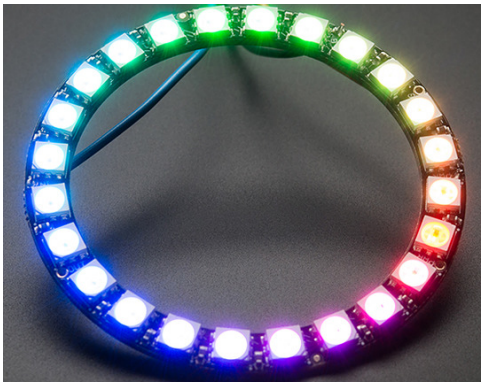


Figure 3.3: 24 LED NeoPixel Ring
[4]

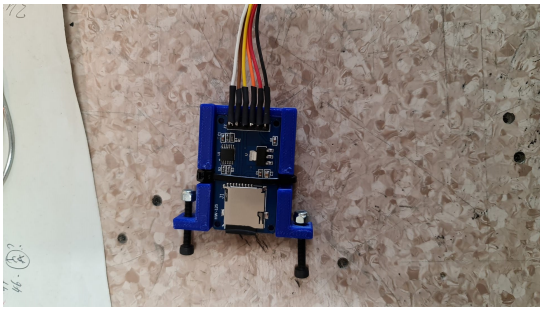
3.1.4. LCD display (HD44780U)

This module serves simply as a more direct means of knowing which windings are inverted. The other peripherals such as the LED ring only display which magnetic poles are incorrect but are unable to tell the user which winding(s) is incorrect. The LCD has the advantage of being able to communicate the incorrect windings in terms of their *line terminal* (i.e. A1, B1, etc), the actual measurement of each magnetic pole and which of these poles do not match the intended magnetic field. The benefit of this LCD device is that it uses the i2c protocol meaning it only needs 2 communication pins to control it and the commands can be heavily simplified though use of a library.

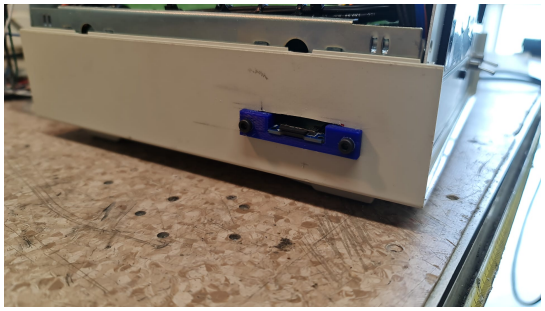
Figure 3.4: Front of machine with LCD display active

3.1.5. SD card Reader/Writer

This module was proposed later on in the project as a means to avoid having to check any of the above peripherals for feedback after each measurement. The idea is to record each measurement directly to an SD card while carefully noting which motors are tested in which order. In this way the operator can set the external computer to run and switch the motor after each test. Then when all the tests are complete, the operator can examine all of the results in a single csv file allowing for ease of automation. The model chosen uses the SPI protocol meaning it requires 4 communication pins to operate however, it is extremely quick to read and write files.



(a) SD Card reader reader



(b) Mounted SD Card reader

3.1.6. 4 pin plug

This acts only as a means for an external computer to control the machine without the need for a human operator. It is conceivable to use a robotic system to automate the testing of a large number of motors. The system could place a new motor on the testing mount, then activate the machine to run the test and record the output and repeat this for each motor. Only 2 of these pins are required for the project.

1. The so called ”error pin” communicates the outcome of a test. It is connected to one of the relays and another pin on the plug. It can signify that the motor has an error or is fine by closing or opening the relay.
2. The activation pin used to control the machine. Refer to the section 3.1.8 to understand how it is used.

Figure 3.6: Example of 4 pin plug [10]

3.1.7. Relays

There are two relays [1] selected work as electronically controlled switches essentially. The first one is used to closed the circuit between the power supply and the terminals of the motor. This relay is only on during the measurement itself so as to avoid power loss and overheating. The second one acts as a signal relay to the external computer. Attached to the external 4 pin connector, it closes the circuit between two pins when an error is detected and leaves it open when the motor is properly wound. The purpose of this is to avoid having to use the system designed in this project as much as possible so that some form of automation can be used. This can not only speed up the testing of motors, but eliminate the operator's need to check both the system and the external computer for each system which reduces the overall amount probability that mistakes may occur.

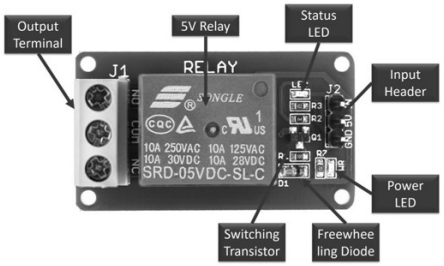


Figure 3.7: 5 V relay
[1]

3.1.8. Switches

These are nothing more than normal 3 channel flip switches which connect electrically the middle pin with either the left or right pin.

Figure 3.8: Front of machine showing switches on the right

The only direct input available to the operator is two switches. A switch on the right of the panel simply turns on the machine. Once the machine is ready and has indicated this to the operator, the switch on the left can be used to carry out each motor test. It is worth noting that the external signal connection for the computer is directly attached to this switch and hence either can be used in the same way to control the machine.

Usage

1. When the switch is pulled HIGH, the machine will carry out a single test of the motor and then await another signal.
2. When the switch is pulled LOW, the error connector will be reset and the machine will be ready for another test.

3.2. Mounting/usage

It is a fairly simple machine and only requires a few things to be done in order to prepare the machine for operation:

- Plug in the motor mount to the 12 pin connector on the back
- Plug in the LED ring to the 4 pin connector on the left of the back
- (Optional) Plug in the external computer to the 4 pin connector on the right of the back
- (Optional) Insert micro SD card into SD card reader
- Place motor on testing mount
- Operate as directed in section 3.1.8

(a) Testing mount with LED ring assembled on top

(b) Testing mount with motor mounted

3.3. Possible improvements

- **Use a pcb.** Not only would this dramatically reduce the size of the machine but the level of capacitance and noise. Internally, there are a lot of wires crossing currently which could easily be avoided with a pcb.
- **Use shorter wires.** The machine is undergoing magnetic interference each time the motor is turned on due to the high amount of current passing through the wires. If these wires can be minimised and placed as far away from the electronics in the machine as possible, then the interference can be reduced.
- **Use a galvanically isolated DC to DC converter.** As it currently stands, the machine uses two separate power supplies to avoid noise caused when the current jolts when the motor turns on. An isolated converter could reduce the size of the machine, as well as costs.

4

Software

The code of this project was designed to be first and foremost effective and efficient, however it was intentionally designed to be understandable for those that aren't very experienced in programming. For these purposes, the program was split into distinct sections.

4.1. Arduino

As a brief background, an Arduino micro-controller can be programmed in various languages using various compilers that serve all types of different purposes. A practical way to develop such a project is to use the provided Arduino IDE as it requires little setup and configuration in order to begin prototyping. The IDE also employs a programming language that resembles greatly C++ just with added functions, keywords, constants, etc. As such this was selected to be the basis of the project.

The *Arduino* language has a very specific template where it always includes Arduino.h and contains an init() function and a loop() function as shown below.

```
1 #include <Arduino.h>
2
3 void setup() {
4   // put your setup code here, to run once:
5
6 }
7
8 void loop() {
9   // put your main code here, to run repeatedly:
10
11 }
```

The setup() function runs once at the start-up of the micro-controller, then loop() runs infinitely as implies its name until the micro-controller is stopped. This is important as setup() will act as a initialisation function for the system and the loop will act as a the main functionality of the device.

4.2. Code structure

The code uses the aforementioned structure as a base which is implemented in *main.ino*. In order to improve readability the project has been separated into separate files that have distinct purposes. *constants.h* contains the global variables whereas *pitches.h* contains only the frequencies of the buzzer, and all functions pertaining to the external peripherals are defined in *peripherals.h* and *peripherals.cpp*.

4.3. Dependencies

In order to use this code the following libraries must be installed onto the Arduino IDE:

- LiquidCrystal_I2C.h [3]

- Adafruit_NeoPixel.h [4]
- Wire.h (built in)
- SD.h (built in)
- SPI.h (built in)

4.4. *main.ino*

Description

Without entering into too much detail, this file acts as the main function for the Arduino Mega. It only defines two functions:

```
1 void initActivationPin(int enablePin)
2 void enableSignalISR(void)
```

void initActivationPin(int enablePin)
Enables the interrupt on either a falling or rising flank for any given interrupt compatible pin while enabling its the pullup resistor.

void enableSignalISR()
Acts as a way of modifying a volatile boolean variable called *activated*. The purpose of this is to prevent the system from acting unless the state of the input pin is changed. The benefit of this is that the the system only has to check that this variable is **true** or not in order become active. There is no polling of the pins required, as a change of state of the pin will set off the interrupt. In this way, the system is free to do other tasks if required and the system reacts immediately because it avoids having to constantly call the function readDigital() which takes a long time to execute.

Function

As stated above, the system is designed in a way to minimise the time spent checking the state of the activation pin. As such, the activation pin has an external interrupt enabled on it whenever there is a **changing flank in the signal**. This is because when the activation pin is pulled:

1. **Down**, the motor connected will be evaluated and the state of error pin will reflect the outcome of this measurement
2. **Up**, the error pin on the plug will be opened to tell the external computer that it is ready for another measurement

Features

At several points a short delay is used to filter out any bouncing in the signal. Because the change in current can be so extreme, the whole system is effected by the resulting oscillations of while it tries to stabilise itself. These delays therefore avoid requiring filtering as they give time for the system to stabilise. An integral example of this the delay of 50 ms at line 33. This comes straight after the program detects the changing flank on the activation pin. At this point the program is aware that the value has changed but not what value it has changed to. By employing a delay, any residual bouncing is skipped and the system can determine whether the activation is now LOW or HIGH. Another fundamental example of this is in line 73. The delay of only 10 ms is enough to prevent the system from completely failing. Without this delay, the system would activate itself each time the motor switched off. The large difference in current supplied would cause the voltage to oscillate radically for a short period which would be interpreted as a changing flank on the activation pin, which would in turn cause the motor to be switched on. This would continue forever, rendering the system unusable.

The program also improves upon the efficiency of the previous one by restricting the time the motor is on. Because it consumes such large amount of power, it is preferable to have the motor for as little time as possible. To achieve this, the motor is turned on, the program waits for 1.2 seconds while the motor is supplied current, then the measurement is taken, and finally the motor is turned off. Lines 36 to 41 correspond to this feature.

To avoid repeating calculations and wasting time re-displaying results, the system also compares the current measurement with the previous in line 45. When it is detected that they are the same, the program uses the much simpler functions *displaySameReadingLCD()* and *displaySameReadingLED()* to communicate this fact. In the case where they aren't the same, the program uses the more much complex logic and display functions to compute and communicate the new results.

4.5. *constants.h*

The file 4.5 contains only definitions of what would otherwise be magic numbers however it also has the added benefit of increasing the flexibility of the code. This is because the pin numbers for each peripheral are defined here. Not only this, the name of the file in which the data will be stored in the micro SD is defined as well as the header of the resulting CSV file.

In this way each constant can be changed depending entirely on the layout of the physical system in addition to the micro-controller used. The beauty of this approach is that as long as a micro-controller is supported by Arduino IDE and has the right amount of pins with the required properties, *constants.h* can be altered to work with a large number of micro-controllers and configurations.

```
1 #ifndef CONSTANTS_H
2 #define CONSTANTS_H
3
4 #define CORRECT 0
5
6 #define UPARROW 0
7 #define DOWNARROW 1
8 #define CROSS 2
9 #define TICK 3
10
11 #define ENABLE_PIN 19
12 #define IN2 12 // tiene que cerrar que hay error
13 #define IN1 13 // tiene que abrir cuando ha terminado de medir (circula el
       corriente)
14
15 #define FIRST_HALL_SENSOR_PIN 22
16 #define NB_HALL_SENSORS 12
17 #define LAST_HALL_SENSOR_PIN 44
18
19 #define LCD_POWER 8
20
21 #define LED_PIN 10
22 #define NB_LEDS 12
23 #define BRIGHTNESS 20
24
25 #define BUZZER_PIN 9
26
27 #define CS_SD 53
28 #define FILENAME "log" //.csv file
29 #define HEADER "No.,Time,Correct(BIN),Measurement(BIN),Correct poles(BIN),
      Winding type"
30 #endif
```

4.6. *pitches.h*

In order to quickly debug and implement the tones corresponding to a correctly manufactured motor and a poorly manufactured one, *pitches.h* was included. The file is directly copied from a GitHub repository [9]. Each definition is simply a note as one would see in the sheet music which is corresponded to its frequency.

4.7. *peripherals.h*

This is simply the header file for *peripherals.cpp* and the functions will hence be described under the section for *peripherals.cpp*. The only noteworthy part of it is the separation of each function prototype. They have been separated by the point at which they are executed in the code. This separation is done namely by whether they are executed during void init() or void loop().

4.8. *peripherals.cpp*

Description

All functions pertaining to the external peripherals are defined here and as such this is the most important file. Each set of functions are grouped by the peripheral they are designed for as well as whether they are in the init or loop function. It also contains some variables which are used for the LCD and for the evaluation of the motor.

Variables

Firstly, the objects that are used to control the LCD and LEDs are instantiated based on default values as well as the configurations in found in *constants.h*. These are used throughout the entire program and heavily simplify the control of these peripherals. Secondly, in order to avoid using complex data structures, having to import large C++ libraries and inefficient memory usage, two arrays are defined. Each of the elements these arrays correspond to each other by index.

The variables are:

1. **uint16_t combinations[]**: which contains the total possible combination of local magnetic poles for a motor with 12 windings (table B.1).
2. **String combinationNames[]**: which contains the string that corresponds to the above combination. In general, it indicates which terminal(s) improperly wound.

Therefore the first element of **combinations[]** corresponds to the first element of **combinationNames[]** and the second with the second and so forth. This property will be used by the program often as a single index can be used to retrieve both of the appropriate elements. A noteworthy feature of this is that it avoids creating a copy of these two arrays until necessary. The index of the combination and the combination name always correspond to each other, so it is the index that passed around the program.

4.9. *peripherals.cpp*

Description

This is the most complex of the files and contains the functions and some definitions that are used to control the external peripherals. For the sake of simplicity, this report will only give brief summaries of functions and some fuller ones for the more important or difficult functions. If there are any doubts the reader can refer to the *GitHub repository*.

Variables

In order to use the more complex external libraries (LCD and LED Ring), they have been instantiated as objects which allows for the low level code to be hidden. *nbMeasurements* also acts to record how many measurements have been made which is used for the data logging in the csv file. *nbCombinations* represents how many possible local magnetic field combinations exist so as to limit loops when searching for a corresponding combination.

Listing 4.1: Variables in peripherals.cpp

```
1 LiquidCrystal_I2C LCD(0x27, 20, 4);
2 Adafruit_NeoPixel LEDstrip(NB_LEDS, LED_PIN, NEO_GRB + NEO_KHZ800);
3 uint16_t nbMeasurements = 0;
4 uint16_t nbCombinations = LENGTH(combinations);
```

As a means to make the machine easier to understand intuitively, the symbols $\uparrow, \downarrow, \checkmark$ and $\times$ and were selected. Unfortunately, these custom characters weren't provided by the LCD's internal memory of characters. They were therefore introduced manually in this file. The up and down arrows will be

used to represent a north polarity and south polarity respectively. And the crossmark and tickmark will represent an incorrect and correct polarity respectively.

Listing 4.2: Custom character definitions in peripherals.cpp

```
1 byte upArrow[];
2 byte downArrow[];
3 byte cross[];
4 byte tick[];
```

These arrays all represent the custom characters, where 1 is a pixel that is on and 0 is a pixel that is off.

Initialisation functions

void initRelays(void)
This just prepares the two relays described in section 3.1.7. Both are set to be initially open and have pullup resistors enabled to avoid ambiguous cases.

void initLCD(void)
The transistor connected between VCC of the LCD and the 5 V of the Arduino is closed. The program then waits briefly for the LCD to turn on and then custom characters are uploaded to the LCD.

void initMessageLCD(bool SDWorking, String fileName)
The inital message is written to the LCD with the exception of the last line. It either writes ** *SD error* ** if the micro SD card (i.e. SDWorking is false) isn't functioning otherwise it prints the name of the file to which the data will be written.

void initHallSensors(void)
The pins controlling the Hall sensors have their pullup resistors enabled.

void initLEDs(void)
The LED rings are turned on and their initial colour is set to white. Their brightness is controlled by the definition *BRIGHTNESS*.

void initBuzzer(void)
The buzzer is enabled by setting its signal pin to *OUTPUT*.

String checkFileNamesSD(void)
The function of this to search the micro SD for existing files. While the base name of the file will always be *FILENAME* (which is "log" in this case), the function will simply increment the number following its name. Simply put, if "log.csv", "log0.csv" and "log1.csv" exist on the micro SD card, then the function will create a new file called "log2.csv" and return its name.

bool initSD(String fileName)
The purpose of this function is to check that the SD card reader and the micro SD card are working as well as print the header or columns of the csv file into the selected file. If both are working then true is returned, otherwise false.

Loop functions

void motorOn(void), void motorOff(void)
These functions simply close or open respectively the relay connected to the motor to allow current to pass through or not.

void errorPortOn(void), void errorPortOff(void)
This is the same as section 4.9 except it controls the error port connected to the external computer.

uint16_t readHallSensors(void)
Here the signal pin of each Hall sensor is read and put into a single 16-bit number where bits $0 \rightarrow 11$ correspond to the starting magnetic pole to the final one. Bit shifting allows the information to be stored in a single variable which would otherwise require 12, 8-bit (=96bits) variables.

uint16_t evaluateMotor(uint16_t reading)
This is the key function of the program. As the array *combinations[]* contains the magnetic poles in the same manner as the function *readHallSensors()* 4.9, the program simply has to cycle through each element in *combinations[]* until it finds a match. If a match is found, its index is returned. If not, *nbCombinations* (which is one larger than the largest possible index) is returned in order to signify that a match could not be found.

void displayStartMeasure(void)
This indicates to the user that a measurement is starting, then it turns off the LCD so as to avoid magnetic interference due to high currents while the motor is on.

void displayEndMeasure(void)
The purpose of this function is to turn on the LCD after the motor is switched off again and then reintialise it using the function *initLCD()* 4.9.

void displayNewReadingLCD(uint8_t comboIndex, uint16_t reading)
In the main loop, it is checked whether the current measurement is the same as the previous. This function is called only if they are not the same. It clears the LCD then displays from left to right the magnetic poles starting from bit 0 to 11. Underneath, the function checks which poles are correct using a bitwise XNOR.

void displayNewReadingLED(uint8_t comboIndex, uint16_t reading)
This works similarly to the above function however it uses green next to the physical position of the winding to signify that it is correct and red when it is wrong. If no combination matches then they are all set to blue.

void displaySameReadingLCD(void)
When the main loop determines that the measurement is the same as the last, this function is called. It simply flashes the LEDs a few times to show the operator that the motor is the same as the previous.

void displaySameReadingLED(uint8_t comboIndex, uint16_t reading)
The functions simply blinks *Same as previous* twice on the last line of the LCD.

void buzzerCorrect(void), void buzzerIncorrect(void)
These functions play the predetermined melodies that signify that the motor is correctly or incorrectly wound.

void writeToSD(String fileName, unsigned long t, uint8_t comboIndex, uint16_t reading)
While appearing somewhat complex, this function simply writes the data of the measurement to the csv file determined at the beginning of the program's execution. It writes the following data to a line in the file:

1. The number of the measurement
2. The time at which the measurement was taken Whether the motor was correct or not
3. The measurement of the poles themselves
4. Which poles were correct and incorrect (represented in binary)
5. The winding type

Special function

boolean getBit(uint16_t b, uint8_t n)
The purpose of this is the retrieve the value of the nth bit in a 16-bit integer called b. It is used in other functions to reduce the length of the code and improve readability.

4.10. Possible improvements

- **Remove the arrays containing the magnetic field combinations and their names.** Instead of storing the combinations an combination names, it could be possible to store only the correct combination and deduce the rest algorithmically. This could reduce memory requirements however it may slow down the system. It would make it much easier to input new motor types however. All that is required to this is to know that each pair of poles in the table B.1 correspond to a winding terminal. For example, *P1* and *P2* correspond to winding A1. Therefore if their polarities are reversed then A1 must be inverted. The same can said for the other terminals. The following poles correspond to the following terminals:

  1. *P1* & *P2* -> A1
  2. *P3* & *P4* -> C2
  3. *P5* & *P6* -> B2
  4. *P7* & *P8* -> A2
  5. *P9* & *P10* -> C1
  6. *P11* & *P12* -> B1

- **Better optimise the delay times.** Most of the time, the delays in each function have a margin added to them to be sure that the system doesn't break. After a fair amount of testing the smallest necessary delay time could be determined. This would improve the program's speed. Not only this but the time that the motor is active could be considerably reduced which could save a lot of power and wear over a long time. This was not reduced as this project's main requirement was an accurate machine and as such extra time was given for the motor's magnetic fields to stablise before taking measurements.
- **Use multi-threading.** The code currently only runs on a single thread meaning that while it is easy to read, the system is stuck doing nothing each time a delay is called. For example, when the LCD is blinking there are short delays between each blink. During this entire time the microcontroller can do nothing else, whereas multi-threading would allow it to do other things in these pauses.

5

Conclusion

A project like this presented a very interesting exploration into real world applications of engineering. The project originally appeared quite straight forward but it took a lot of trial and error to get the system to run as intended. It took even more work to then optimise the program however by then more problems would arise. What was interesting was the strange line of seemingly unrelated problems that each required unique solutions.

The project is far from being done but it can serve as a good base for the next person. A little more time would have served well to really correct any remaining issues and improve robustness as well as scalability. As it currently stands, the machine is able to accept different configurations of 3 phase motors however, to implement them is not as simple as inputting the magnetic poles. It is possible but it would either require a good understanding of the code or my assistance.

I'd like to specifically thank professor Ramon Bargallo for giving me the chance to work with him at such late notice. He was under no obligation but worked with me to have a project that was achievable in such a short amount of time. His sustained support throughout the entire project inspired me to work hard as well as explore all the possible sources of problems thanks to his expertise in electric motors.

I'd also like to thank the lab assistant Jordi Vilanova for his help in assembling and improving the system week by week. It was very helpful to be able to discuss possible solutions with him and then have him implement ones we could agree on.

References

[1] *5V relay module : Pin Configuration, circuit, working & its applications*. Aug. 2021. URL: https://www.elprocus.com/5v-relay-module/.

[2] *Arduino Buzzer*. 2022. URL: https://es.aliexpress.com/item/32525682460.html?gatewayAdapt=glo2esp.

[3] Frank de Brabander. *Arduino-LiquidCrystal-I2C-library*. 2022. URL: https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library.

[4] Phil Burgess. *Adafruit_NeoPixel*. 2022. URL: https://github.com/adafruit/Adafruit_NeoPixel.

[5] *Digital Output Hall Effect Sensor*. 2018. URL: https://microcontrollerslab.com/hall-effect-sensor-working/.

[6] Ed Edwards. *What is a 3-phase motor and how does it work?* URL: https://www.thomasnet.com/articles/machinery-tools-supplies/what-is-a-3-phase-motor-and-how-does-it-work/.

[7] Adafruit Industries. *Neopixel ring - 24 x 5050 RGB led with integrated drivers*. URL: https://www.adafruit.com/product/1586.

[8] *Lenz's law*. June 2022. URL: https://en.wikipedia.org/wiki/Lenz's_Law.

[9] Mike Putnam. *pitches.h*. 2022. URL: https://gist.github.com/mikeputnam/2820675.

[10] *Standard Industrial Cable M12 Connector 4 pin aviation plug*. Nov. 2018. URL: https://adamconn.com/product/standard-industrial-cable-m12-connector-4-pin-aviation-plug.

A

Source Code

This contains the current source code of the project. It is preferable to use the GitHub repository as it is guaranteed to be up to date, however this is included for redundancy.

Listing A.1: *main.ino*

```
#include "peripherals.h"
#include "constants.h"
//#define DEBUG

volatile boolean activated = false;

void enableSignalISR(void);
void initActivationPin(int enablePin);

String fileName = "";
bool SDWorking = false;

void setup() {
  Serial.begin(115200);
  delay(50);

  //initialise IO peripherals
  fileName = checkFileNamesSD();
  SDWorking = initSD(fileName);
  initLCD();
  initMessageLCD(SDWorking, fileName);
  initHallSensors();
  initLEDs();
  initRelays();
  initActivationPin(ENABLE_PIN);
}

uint16_t oldMeasurement = 0;
uint8_t comboIndex = nbCombinations;

void loop() {
  if (activated) {
    delay(50); //Check that signal really is LOW
    if(digitalRead(ENABLE_PIN) == LOW){ //DOWNWARD Flank
      displayStartMeasure();
      motorOn(); //Allow current to flow to create magnetic field
```

```
37        delay(1200); //Wait for relays to turn on before making a
              measurement, could be longer (than 1000) because it doesn't reach
               full current yet. But the measurements look good
38
39        unsigned long measureTime = millis();
40        uint16_t measurement = readHallSensors();
41        motorOff(); //Deactivate motor to conserve power
42        delay(100);
43        displayEndMeasure();
44
45        if(measurement != oldMeasurement){ //Check if current measurement is
               the same as previous
46          comboIndex = evaluateMotor(measurement);
47          displayNewReadingLCD(comboIndex, measurement);
48          displayNewReadingLED(comboIndex, measurement);
49
50          oldMeasurement=measurement;
51        }
52        else{
53          displaySameReadingLCD(comboIndex, oldMeasurement);
54          displaySameReadingLED();
55        }
56        if(comboIndex == CORRECT){ // check if the motor is correct or not
57          buzzerCorrect();
58          errorPortOff(); //Not necessary but good for redundancy
59        }
60        else{
61          buzzerIncorrect();
62          errorPortOn();
63        }
64
65        if(SDWorking)
66          writeToSD(fileName, measureTime, comboIndex, measurement);
67      }
68      else{
69        delay(50); //Debounce
70        if(digitalRead(ENABLE_PIN) == HIGH) //UPWARD Flank
71          errorPortOff(); //reset error pin
72      }
73      delay(10); // When the relays turn off, they create a bounce in the
            signal which activates measurement again and never stops: wait for
            bounce then continue
74      activated = false;
75    }
76  }
77
78  void enableSignalISR(void) {
79    activated = true;
80  }
81
82  void initActivationPin(int enablePin){
83    pinMode(enablePin, INPUT_PULLUP); //Init activation signal pin
84    attachInterrupt(digitalPinToInterrupt(enablePin), enableSignalISR,
          CHANGE); //interrupt for the enable signal
85  }
```

Listing A.2: *constants.h*

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#define CORRECT 0

#define UPARROW 0
#define DOWNARROW 1
#define CROSS 2
#define TICK 3

#define ENABLE_PIN 19
#define IN2 12 // tiene que cerrar que hay error
#define IN1 13 // tiene que abrir cuando ha terminado de medir (circula el corriente)

#define FIRST_HALL_SENSOR_PIN 22
#define NB_HALL_SENSORS 12
#define LAST_HALL_SENSOR_PIN 44

#define LCD_POWER 8

#define LED_PIN 10
#define NB_LEDS 12
#define BRIGHTNESS 20

#define BUZZER_PIN 9

#define CS_SD 53
#define FILENAME "log" //.csv file
#define HEADER "No.,Time,Correct(BIN),Measurement(BIN),Correct poles(BIN),Winding type"
#endif

//Interruptor abajo (a HIGH) -> abre IN2
// Memoria el 20 Lunes
```

Listing A.3: *peripherals.h*

```
#ifndef PERIPHERALS_H
#define PERIPHERALS_H

#include "constants.h"
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <Adafruit_NeoPixel.h>
#include <SPI.h>
#include <SD.h>

inline bool getBit(uint16_t b, uint8_t n) __attribute__((always_inline)); //inline in order to improve speed

//Template/Macro for counting number of elements in array safely, works for many types
template <typename T, size_t N>
char (&_ArraySizeHelper( T (&arr)[N] ))[N];
#define LENGTH( arr ) (sizeof( _ArraySizeHelper( arr ) ))
```

```
17
18 extern uint16_t nbCombinations;
19
20 /******Initialisation functions*******/
21 void initRelays(void);
22 String checkFileNamesSD(void);
23 bool initSD(String fileName);
24 void initLCD(void);
25 void initMessageLCD(bool SDWorking, String fileName);
26 void initHallSensors(void);
27 void initLEDs(void);
28 void initBuzzer(void);
29 /******Initialisation functions*******/
30
31 /******Loop functions******/
32 void motorOn(void);
33 void motorOff(void);
34 void errorPortOn(void);
35 void errorPortOff(void);
36
37 uint16_t readHallSensors(void);
38 uint16_t evaluateMotor(uint16_t reading);
39
40 void displayStartMeasure(void);
41 void displayEndMeasure(void);
42 void displayNewReadingLCD(uint8_t comboIndex, uint16_t reading);
43 void displayNewReadingLED(uint8_t comboIndex, uint16_t reading);
44
45 void displaySameReadingLCD(uint8_t comboIndex, uint16_t reading); //blink
    screen with "Same as previous"
46 void displaySameReadingLED(void);
47
48 void buzzerCorrect(void);
49 void buzzerIncorrect(void);
50
51 void writeToSD(String fileName, unsigned long t, uint8_t comboIndex,
    uint16_t reading);
52 /******Loop functions******/
53
54 #endif
```

Listing A.4: *peripherals.cpp*

```
1 #include "peripherals.h"
2 #include "constants.h"
3 #include "pitches.h"
4
5 LiquidCrystal_I2C lcd(0x27, 20, 4); // set the LCD address to 0x27 for a
    16 chars and 2 line
6 Adafruit_NeoPixel LEDstrip(NB_LEDS, LED_PIN, NEO_GRB + NEO_KHZ800); //12 =
    NB_HALL_SENSORS
7 uint16_t nbMeasurements = 0;
8
9 /*******CONSTANTS*******/
10 //Known polarity combinations
11 //Each bit represents a polarity: 1-> N pole, 0-> S pole
12 //LSb represents the measurement of the last hall sensor =
```

```
13 uint16_t combinations[] = {
14   0b101001010110, 0b011001010110, 0b101001100110, 0b011001100110,
       0b101001010101,
15   0b011001010101, 0b101001100101, 0b011001100101, 0b101010010110,
       0b011010010110,
16   0b101010100110, 0b011010100110, 0b101010010101, 0b011010010101,
       0b101010100101,
17   0b011010100101, 0b101001011010, 0b011001011010, 0b101001101010,
       0b011001101010,
18   0b101001011001, 0b011001011001, 0b101001101001, 0b011001101001,
       0b101010011010,
19   0b011010011010, 0b101010101010, 0b011010101010, 0b101010011001,
       0b011010011001,
20   0b101010101001, 0b011010101001, 0b100101010110, 0b010101010110,
       0b100101100110,
21   0b010101100110, 0b100101010101, 0b010101010101, 0b100101100101,
       0b010101100101,
22   0b100110010110, 0b010110010110, 0b100110100110, 0b010110100110,
       0b100110010101,
23   0b010110010101, 0b100110100101, 0b010110100101, 0b100101011010,
       0b010101011010,
24   0b100101101010, 0b010101101010, 0b100101011001, 0b010101011001,
       0b100101101001,
25   0b010101101001, 0b100110011010, 0b010110011010, 0b100110101010,
       0b010110101010,
26   0b100110011001, 0b010110011001, 0b100110101001, 0b010110101001
27 };
28
29 //Corresponding code for each polarity combination
30 String combinationNames[] = {
31   "Correct", "A1inv", "A2inv", "Ainv", "B1inv",
32   "A1B1inv", "A2B1inv", "AB1inv", "B2inv", "A1B2inv",
33   "A2B2inv", "AB2inv", "Binv", "A1Binv", "A2Binv",
34   "ABinv", "C1inv", "A1C1inv", "A2C1inv", "AC1inv",
35   "B1C1inv", "A1B1C1inv", "A2B1C1inv", "AB1C1inv", "B2C1inv",
36   "A1B2C1inv", "A2B2C1inv", "AB2C1inv", "BC1inv", "A1BC1inv",
37   "A2BC1inv", "ABC1inv", "C2inv", "A1C2inv", "A2C2inv",
38   "AC2inv", "B1C2inv", "A1B1C2inv", "A2B1C2inv", "AB1C2inv",
39   "B2C2inv", "A1B2C2inv", "A2B2C2inv", "AB2C2inv", "BC2inv",
40   "A1BC2inv", "A2BC2inv", "ABC2inv", "Cinv", "A1Cinv",
41   "A2Cinv", "ACinv", "B1Cinv", "A1B1Cinv", "A2B1Cinv",
42   "AB1Cinv", "B2Cinv", "A1B2Cinv", "A2B2Cinv", "AB2Cinv",
43   "BCinv", "A1BCinv", "A2BCinv", "ABCinv", "Not found"
44 };
45
46 byte upArrow[] = {
47   0b00000,
48   0b00100,
49   0b01110,
50   0b10101,
51   0b00100,
52   0b00100,
53   0b00100,
54   0b00000
55 };
56
```

```
57 byte downArrow[] = {
58   0b00000,
59   0b00100,
60   0b00100,
61   0b00100,
62   0b10101,
63   0b01110,
64   0b00100,
65   0b00000
66 };
67
68 byte cross[] = {
69   0b00000,
70   0b10001,
71   0b01010,
72   0b00100,
73   0b01010,
74   0b10001,
75   0b00000,
76   0b00000
77 };
78
79 byte tick[] = {
80   0b00000,
81   0b00000,
82   0b00000,
83   0b00001,
84   0b00010,
85   0b10100,
86   0b01000,
87   0b00000
88 };
89
90 uint16_t nbCombinations = LENGTH(combinations); //depends only on array '
       combinations', therefore adaptable for different motor configurations
91 /*******CONSTANTS*******/
92
93 /******Initialisation functions*******/
94 void initRelays(void){
95   pinMode(IN1, OUTPUT);
96   pinMode(IN2, OUTPUT);
97   motorOff();
98   errorPortOff();
99 }
100
101 void initLCD(void){
102   pinMode(LCD_POWER, INPUT_PULLUP);
103   digitalWrite(LCD_POWER, HIGH);
104   delay(100);
105
106   lcd.init();
107   lcd.backlight();
108   lcd.createChar(UPARROW, upArrow);
109   lcd.createChar(DOWNARROW, downArrow);
110   lcd.createChar(CROSS, cross);
111   lcd.createChar(TICK, tick);
```

```
112 }
113
114 void initMessageLCD(bool SDWorking, String fileName){
115   lcd.print("State:no measurement");
116   lcd.setCursor(0, 1);
117   lcd.print("Awaiting activation");
118   lcd.setCursor(0, 3);
119   if(!SDWorking){
120     lcd.print("   ** SD error **");
121   }
122   else{
123     lcd.print("File: " + fileName);
124   }
125 }
126
127 void initHallSensors(void) {
128   for (int i = FIRST_HALL_SENSOR_PIN; i <= LAST_HALL_SENSOR_PIN /*
        NB_HALL_SENSORS + FIRST_HALL_SENSOR_PIN*/; i+=2) { //MSB first
129  #ifdef DEBUG
130     Serial.println(i);
131  #endif
132     pinMode(i, INPUT_PULLUP);
133   }
134 }
135
136 void initLEDs(void){
137   LEDstrip.begin();
138   LEDstrip.setBrightness(BRIGHTNESS);
139   LEDstrip.fill(LEDstrip.Color(255, 255, 255)); //White
140   LEDstrip.setPixelColor(0, LEDstrip.Color(0, 117, 255)); //Set first
        pixel to Blue to show start
141   LEDstrip.show();
142 }
143
144 void initBuzzer(void){
145   pinMode(BUZZER_PIN, OUTPUT);
146 }
147
148 String checkFileNamesSD(void){
149   if (!SD.begin(CS_SD)) {
150     Serial.println("SD card or not present.");
151     return ""; // don't do anything more:
152   }
153
154   String fileName = FILENAME;
155   int len = fileName.length();
156   if(!SD.exists(fileName + ".csv"))
157     return fileName + ".csv";
158
159   uint8_t i = 0;
160   while(1){
161     if(!SD.exists(fileName + i + ".csv"))
162       return fileName + i + ".csv"; //i is incremented if a file of the
            same name is detected
163     i++;
164   }
```

```
165 }
166
167 bool initSD(String fileName){
168   if (!SD.begin(CS_SD)) {
169     Serial.println("SD card broken or not present.");
170     return false; // don't do anything more:
171   }
172
173   File logFile = SD.open(fileName, FILE_WRITE);
174   if (logFile){
175     //These will be the headers for your excel file, CHANGE "" to whatever
          headers you would like to use
176     logFile.println("sep=,");
177     logFile.println(HEADER);
178     logFile.close();
179     return true;
180   }
181   Serial.println("Unable to open.");
182   return false;
183 }
184 /******Initialisation functions*******/
185
186 /******Loop functions******/
187 void motorOn(void){
188   digitalWrite(IN2, LOW); //Closed
189 }
190
191 void motorOff(void){
192   digitalWrite(IN2, HIGH); //Open
193 }
194
195 void errorPortOn(void){
196   digitalWrite(IN1, LOW); //Closed
197 }
198
199 void errorPortOff(void){
200   digitalWrite(IN1, HIGH); //Open
201 }
202
203 uint16_t readHallSensors(void) {
204   uint16_t reading = 0;
205
206   for (int i = FIRST_HALL_SENSOR_PIN, j = NB_HALL_SENSORS - 1 ; i <=
        LAST_HALL_SENSOR_PIN; i+=2, j--) { //MSB first
207     reading |= (digitalRead(i) << j); // each reading is a single bit, the
          bit shifting enables this
208   }
209 #ifdef DEBUG
210   Serial.print("reading = ");
211   Serial.println(reading, BIN);
212 #endif
213   return reading;
214 }
215
216 uint16_t evaluateMotor(uint16_t reading) {
217   for (int i = 0; i < nbCombinations - 1; i++) {
```

```
218 #ifdef DEBUG
219     Serial.print(reading, BIN);
220     Serial.print(" =? ");
221     Serial.println((combinations[i]), BIN);
222 #endif
223     if (reading == combinations[i]) {
224       //strcpy(s, combinationNames[i].c_str());
225       return i; //index of correct combination
226 #ifdef DEBUG
227       Serial.println(s);
228 #endif
229     }
230   }
231
232 #ifdef DEBUG
233   Serial.println(s);
234 #endif
235   return nbCombinations; //No correponding combination found
236 }
237
238 void displayStartMeasure(void){
239   lcd.setCursor(0, 3);
240   lcd.print("Measuring poles    ");
241   //lcd.noDisplay();
242   digitalWrite(LCD_POWER, LOW);
243 }
244
245 void displayEndMeasure(void){
246   //lcd.display();
247   digitalWrite(LCD_POWER, HIGH);
248   initLCD();
249 }
250
251 void displayNewReadingLCD(uint8_t comboIndex, uint16_t reading) {
252   lcd.clear();
253   delay(10);
254   for(int i = 0; i < 3; i++){ //repeat data sending so that the screen is
          more stable
255     lcd.home();
256     lcd.print("State: ");
257     lcd.print(combinationNames[comboIndex]);
258
259     lcd.setCursor(0, 1);
260     if(comboIndex >= 0){ //print raw pole readings
261       lcd.print("Poles: ");
262       for(int i=NB_HALL_SENSORS-1; i >= 0; i--)
263         getBit(reading, i) ? lcd.write(UPARROW) : lcd.write(DOWNARROW);
264
265       lcd.setCursor(7, 2); //align with polarities
266       reading ^= combinations[CORRECT];
267       for(int i=NB_HALL_SENSORS-1; i >= 0; i--){ //print whether poles are
              correct or not
268         getBit(reading, i) ? lcd.write(CROSS) : lcd.write(TICK); //Negated
              because with xor 1 means the values are different
269       }
270       lcd.setCursor(0, 3);
```

```
271       lcd.print("          ");
272     }
273     else{
274       //lcd.print("- Please try again -");
275     }
276   }
277 }
278
279 void displayNewReadingLED(uint8_t comboIndex, uint16_t reading){
280   reading ^= combinations[CORRECT]; //Find incorrect windings: 1 where the
          values are different ie. wrong
281   LEDstrip.clear();
282   if(comboIndex >=0){
283     for(int i=NB_HALL_SENSORS-1; i >= 0;i--){ //Start from MSb -> LSb
284       uint32_t colour = LEDstrip.Color(0, 255, 0); //Green
285
286       if(getBit(reading, i)) //This checks if the bit at the given index
             is 1 or 0
287         colour = LEDstrip.Color(255, 0, 0); //Red
288
289       LEDstrip.setPixelColor(i, colour);
290       LEDstrip.show();
291       delay(35);
292     }
293   }
294   else{
295     LEDstrip.fill(LEDstrip.Color(0, 0, 255)); //Blue
296     LEDstrip.show();
297   }
298 }
299
300 void displaySameReadingLCD(uint8_t comboIndex, uint16_t reading){
301     displayNewReadingLCD(comboIndex, reading);
302     lcd.setCursor(0, 3);
303     lcd.print("Same as previous");
304     delay(500);
305     lcd.setCursor(0, 3);
306     lcd.print("                    "); //clear line
307     delay(500);
308     lcd.setCursor(0, 3);
309     lcd.print("Same as previous");
310 }
311
312 void displaySameReadingLED(void){
313   for(int j=0; j<5; j++){
314     for(int i=0; i<BRIGHTNESS; i++){ //DIM LEDS
315       LEDstrip.setBrightness(BRIGHTNESS-i);
316       LEDstrip.show();
317       delay(8);
318     }
319
320     delay(8);
321
322     for(int i=1; i<=BRIGHTNESS;i++){ //BRIGHTEN LEDs
323       LEDstrip.setBrightness(i);
324       LEDstrip.show();
```

```
325       delay(8);
326     }
327   }
328 }
329
330 void buzzerCorrect(void){
331   int melodyGood[] = {NOTE_C5, NOTE_E5, NOTE_G5};
332   int duration = 200;
333   for (int thisNote = 0; thisNote < 3; thisNote++) {
334     tone(BUZZER_PIN, melodyGood[thisNote], duration);
335     delay(200);
336   }
337 }
338
339 //void buzzerIncorrect(void){
340 //  int melodyBad[] = {NOTE_CS4, NOTE_C5};
341 //  int duration = 300;
342 //  for(int j = 0; j < 4; j++){
343 //    for (int thisNote = 0; thisNote < 2; thisNote++) {
344 //      tone(BUZZER_PIN, melodyBad[thisNote], duration);
345 //      delay(20);
346 //    }
347 //  }
348 //}
349
350 void buzzerIncorrect(void){
351   int melodyGood[] = {NOTE_G4, NOTE_G4};
352   int duration = 200;
353   for (int thisNote = 0; thisNote < 2; thisNote++) {
354     tone(BUZZER_PIN, melodyGood[thisNote], duration);
355     delay(200);
356   }
357 }
358
359 void writeToSD(String fileName, unsigned long t, uint8_t comboIndex,
      uint16_t reading){
360   File logFile = SD.open(fileName, FILE_WRITE);
361   if (logFile){
362     uint16_t mask = 0b0000111111111111; //remove the first 4 bits of
          measurement which correspond to nothing
363     uint16_t correctPoles = ~(reading ^= combinations[CORRECT]);
364     correctPoles &= mask;
365
366       logFile.print(String(nbMeasurements) + ","
367                     + String(t/1000.0) + ","                          //
                          Time
368                     + String(comboIndex == CORRECT) + ",");           //Is
                          the motor completely correct?
369       logFile.print(reading, BIN);                                    //
            Raw measurement
370       logFile.print(",");
371       logFile.print(correctPoles, BIN);                               //
            Correct/incorrect poles
372       logFile.print(",");
373       logFile.println(combinationNames[comboIndex]);                  //
            Winding type
```

```
374         logFile.close();
375
376         //For debugging purposes
377         String s = (String(nbMeasurements) + ","
378                         + String(comboIndex == CORRECT) + ","
379                         + String(t) + ","
380                         + String(reading) + ","
381                         + String(~(reading ^= combinations[CORRECT])) + ","
382                         + combinationNames[comboIndex]);
383         Serial.println(s);
384         nbMeasurements++;
385       }
386 }
387 /******Loop functions******/
388
389 /******Supplementary function******/
390 boolean getBit(uint16_t b, uint8_t n){
391   uint16_t mask = 1<<n;
392   return b & mask;
393 }
394 /******Supplementary function******/
```

Listing A.5: *pitches.h*

```
1  /*************************************************
2
3   Frequency of notes
4
5   *************************************************/
6  #define NOTE_B0  31
7  #define NOTE_C1  33
8  #define NOTE_CS1 35
9  #define NOTE_D1  37
10 #define NOTE_DS1 39
11 #define NOTE_E1  41
12 #define NOTE_F1  44
13 #define NOTE_FS1 46
14 #define NOTE_G1  49
15 #define NOTE_GS1 52
16 #define NOTE_A1  55
17 #define NOTE_AS1 58
18 #define NOTE_B1  62
19 #define NOTE_C2  65
20 #define NOTE_CS2 69
21 #define NOTE_D2  73
22 #define NOTE_DS2 78
23 #define NOTE_E2  82
24 #define NOTE_F2  87
25 #define NOTE_FS2 93
26 #define NOTE_G2  98
27 #define NOTE_GS2 104
28 #define NOTE_A2  110
29 #define NOTE_AS2 117
30 #define NOTE_B2  123
31 #define NOTE_C3  131
32 #define NOTE_CS3 139
33 #define NOTE_D3  147
```

```
34 #define NOTE_DS3 156
35 #define NOTE_E3  165
36 #define NOTE_F3  175
37 #define NOTE_FS3 185
38 #define NOTE_G3  196
39 #define NOTE_GS3 208
40 #define NOTE_A3  220
41 #define NOTE_AS3 233
42 #define NOTE_B3  247
43 #define NOTE_C4  262
44 #define NOTE_CS4 277
45 #define NOTE_D4  294
46 #define NOTE_DS4 311
47 #define NOTE_E4  330
48 #define NOTE_F4  349
49 #define NOTE_FS4 370
50 #define NOTE_G4  392
51 #define NOTE_GS4 415
52 #define NOTE_A4  440
53 #define NOTE_AS4 466
54 #define NOTE_B4  494
55 #define NOTE_C5  523
56 #define NOTE_CS5 554
57 #define NOTE_D5  587
58 #define NOTE_DS5 622
59 #define NOTE_E5  659
60 #define NOTE_F5  698
61 #define NOTE_FS5 740
62 #define NOTE_G5  784
63 #define NOTE_GS5 831
64 #define NOTE_A5  880
65 #define NOTE_AS5 932
66 #define NOTE_B5  988
67 #define NOTE_C6  1047
68 #define NOTE_CS6 1109
69 #define NOTE_D6  1175
70 #define NOTE_DS6 1245
71 #define NOTE_E6  1319
72 #define NOTE_F6  1397
73 #define NOTE_FS6 1480
74 #define NOTE_G6  1568
75 #define NOTE_GS6 1661
76 #define NOTE_A6  1760
77 #define NOTE_AS6 1865
78 #define NOTE_B6  1976
79 #define NOTE_C7  2093
80 #define NOTE_CS7 2217
81 #define NOTE_D7  2349
82 #define NOTE_DS7 2489
83 #define NOTE_E7  2637
84 #define NOTE_F7  2794
85 #define NOTE_FS7 2960
86 #define NOTE_G7  3136
87 #define NOTE_GS7 3322
88 #define NOTE_A7  3520
89 #define NOTE_AS7 3729
```

```
90 #define NOTE_B7  3951
91 #define NOTE_C8  4186
92 #define NOTE_CS8 4435
93 #define NOTE_D8  4699
94 #define NOTE_DS8 4978
```

B

Binary local magnetic field representations

This table is the result of simulations run by professor Ramon Bargallo. The ”Winding” column shows which windings are inverted. The ”Px” columns show the expected local magnetic polarity for the particular ”Winding” configuration. Only the first row is considered correct.

Table B.1: Possible magnetic polarities and their corresponding winding configurations

Winding	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Correct	1	0	1	0	0	1	0	1	0	1	1	0
A1inv	0	1	1	0	0	1	0	1	0	1	1	0
A2inv	1	0	1	0	0	1	1	0	0	1	1	0
Ainv	0	1	1	0	0	1	1	0	0	1	1	0
B1inv	1	0	1	0	0	1	0	1	0	1	0	1
A1B1inv	0	1	1	0	0	1	0	1	0	1	0	1
A2B1inv	1	0	1	0	0	1	1	0	0	1	0	1
AB1inv	0	1	1	0	0	1	1	0	0	1	0	1
B2inv	1	0	1	0	1	0	0	1	0	1	1	0
A1B2inv	0	1	1	0	1	0	0	1	0	1	1	0
A2B2inv	1	0	1	0	1	0	1	0	0	1	1	0
AB2inv	0	1	1	0	1	0	1	0	0	1	1	0
Binv	1	0	1	0	1	0	0	1	0	1	0	1
A1Binv	0	1	1	0	1	0	0	1	0	1	0	1
A2Binv	1	0	1	0	1	0	1	0	0	1	0	1
ABinv	0	1	1	0	1	0	1	0	0	1	0	1
C1inv	1	0	1	0	0	1	0	1	1	0	1	0
A1C1inv	0	1	1	0	0	1	0	1	1	0	1	0
A2C1inv	1	0	1	0	0	1	1	0	1	0	1	0
AC1inv	0	1	1	0	0	1	1	0	1	0	1	0
B1C1inv	1	0	1	0	0	1	0	1	1	0	0	1
A1B1C1inv	0	1	1	0	0	1	0	1	1	0	0	1
A2B1C1inv	1	0	1	0	0	1	1	0	1	0	0	1
AB1C1inv	0	1	1	0	0	1	1	0	1	0	0	1
B2C1inv	1	0	1	0	1	0	0	1	1	0	1	0
A1B2C1inv	0	1	1	0	1	0	0	1	1	0	1	0
A2B2C1inv	1	0	1	0	1	0	1	0	1	0	1	0
AB2C1inv	0	1	1	0	1	0	1	0	1	0	1	0
BC1inv	1	0	1	0	1	0	0	1	1	0	0	1
A1BC1inv	0	1	1	0	1	0	0	1	1	0	0	1

Table B.1 continued from previous page

A2BC1inv	1	0	1	0	1	0	1	0	1	0	0	1
ABC1inv	0	1	1	0	1	0	1	0	1	0	0	1
C2inv	1	0	0	1	0	1	0	1	0	1	1	0
A1C2inv	0	1	0	1	0	1	0	1	0	1	1	0
A2C2inv	1	0	0	1	0	1	1	0	0	1	1	0
AC2inv	0	1	0	1	0	1	1	0	0	1	1	0
B1C2inv	1	0	0	1	0	1	0	1	0	1	0	1
A1B1C2inv	0	1	0	1	0	1	0	1	0	1	0	1
A2B1C2inv	1	0	0	1	0	1	1	0	0	1	0	1
AB1C2inv	0	1	0	1	0	1	1	0	0	1	0	1
B2C2inv	1	0	0	1	1	0	0	1	0	1	1	0
A1B2C2inv	0	1	0	1	1	0	0	1	0	1	1	0
A2B2C2inv	1	0	0	1	1	0	1	0	0	1	1	0
AB2C2inv	0	1	0	1	1	0	1	0	0	1	1	0
BC2inv	1	0	0	1	1	0	0	1	0	1	0	1
A1BC2inv	0	1	0	1	1	0	0	1	0	1	0	1
A2BC2inv	1	0	0	1	1	0	1	0	0	1	0	1
ABC2inv	0	1	0	1	1	0	1	0	0	1	0	1
Cinv	1	0	0	1	0	1	0	1	1	0	1	0
A1Cinv	0	1	0	1	0	1	0	1	1	0	1	0
A2Cinv	1	0	0	1	0	1	1	0	1	0	1	0
ACinv	0	1	0	1	0	1	1	0	1	0	1	0
B1Cinv	1	0	0	1	0	1	0	1	1	0	0	1
A1B1Cinv	0	1	0	1	0	1	0	1	1	0	0	1
A2B1Cinv	1	0	0	1	0	1	1	0	1	0	0	1
AB1Cinv	0	1	0	1	0	1	1	0	1	0	0	1
B2Cinv	1	0	0	1	1	0	0	1	1	0	1	0
A1B2Cinv	0	1	0	1	1	0	0	1	1	0	1	0
A2B2Cinv	1	0	0	1	1	0	1	0	1	0	1	0
AB2Cinv	0	1	0	1	1	0	1	0	1	0	1	0
BCinv	1	0	0	1	1	0	0	1	1	0	0	1
A1BCinv	0	1	0	1	1	0	0	1	1	0	0	1
A2BCinv	1	0	0	1	1	0	1	0	1	0	0	1
ABCinv	0	1	0	1	1	0	1	0	1	0	0	1

C

Magnetic field simulations

Below are the results of simulations run by Ramon Bargallo of the magnetic field inside a stator when supplied current. The measurement follows a circular path around the stator meaning that the distance axis represents the distance around the circumference.

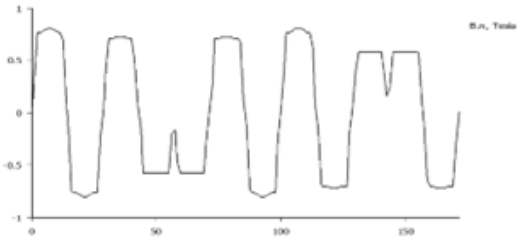


Figure C.1: Motor with correct windings

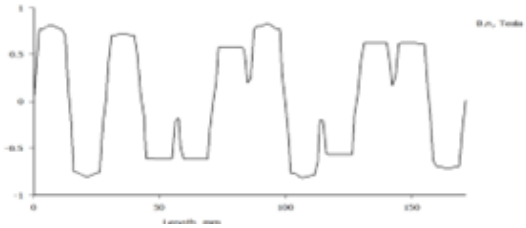


Figure C.2: Motor with winding A inverted

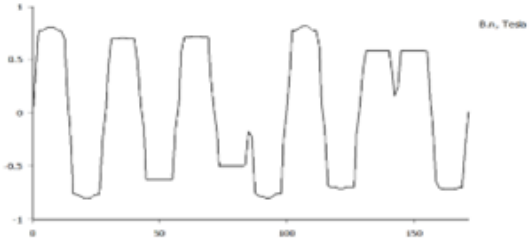


Figure C.3: Motor with winding B inverted

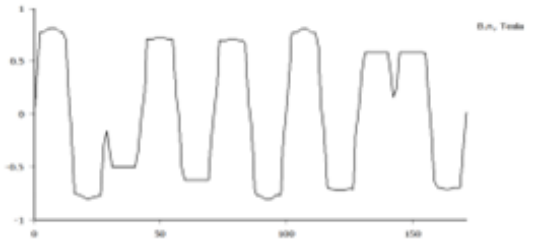


Figure C.4: Motor with winding C inverted

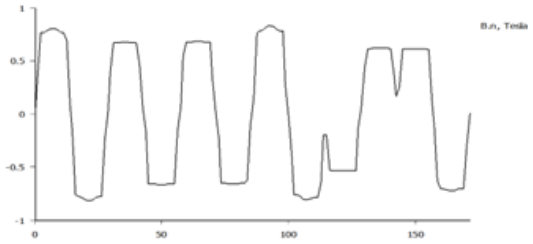


Figure C.5: Motor with winding A and B inverted

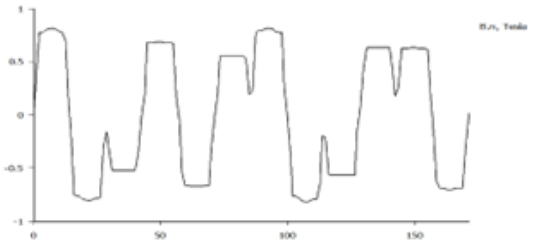


Figure C.6: Motor with winding A and C inverted

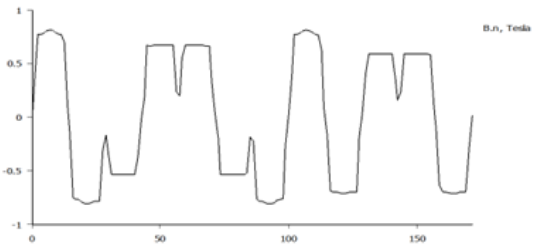


Figure C.7: Motor with winding B and C inverted

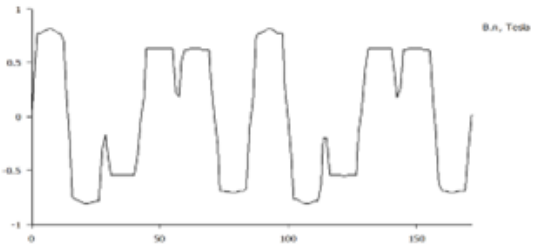


Figure C.8: Motor with winding A, B and C inverted