INTELLIGENT CONTROLLER BASED ON A RASPBERRY PI

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

Giorgos Tsapparellas
Student ID: 10392495

School of Computer Science

Contents

Word Count: 24.676

List of Tables

List of Figures

List of Equations

List of Symbols

G(s)	Transfer function as operator
I(s)	Input of a control system
O(s)	Output of a control system
e(t)	Error signal in respect with time t
τ	Integral interval time
Kp	Proportional gain
Ki	Integral gain
Kd	Derivative gain
Kpu	Ultimate gain
Tu	Ultimate oscillation period
Oe	Oersted

List of Abbreviations

ASCII	American Standard for Information Interchange
D	Derivative term
GPIB	General Purpose Interface Bus
GUI	Graphical User Interface
HSE	High Sensitivity Probe
I	Integral term
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
MF	Magnetic Field
MIMO	Multiple-Input-Multiple-Output
MISO	Multiple-Input-Single-Output
NI	National Instruments
P	Proportional term
PC	Personal Computer
PD	Proportional-Derivative term
PI	Proportional-Integral term
PID	Proportional-Integral-Derivative term
SASO	Stability-Accuracy-Speed-Overshooting performance indices
SCPI	Standard Commands for Programmable Instruments
SIMO	Single-Input-Multiple-Output
SISO	Single-Input-Single-Output
TL	Tyreus-Luyben
UI	User Interface
UML	Unified Modelling Language
UoM	University of Manchester
USB	Universal Serial Bus
ZN	Ziegler-Nichols

Abstract

INTELLIGENT CONTROLLER BASED ON A RASPBERRY PI
Giorgos Tsapparellas
A dissertation submitted to the University of Manchester
for the degree of Master of Science, 2021

Despite the usefulness of the automated mechanical applications, modern industrial society is constantly replacing mechanical methods utilized over the past decades with automated electronic solutions, in the form of intelligent controllers. This is due to the adaptive, reproducible, precise, robust and self-organizing characteristics required by the modern industrial society. Intelligent controllers are omnipresent, performing many day-to-day tasks that we take for granted. Straightforward applications of washing machine and control heating systems, as well as, more advanced applications of fuel pressure, liquid level control and power steering are being assisted by the intelligent controllers. The basic requirements of an intelligent controller include the control of a sensor output together with a set of desired goals, which are used as the control parameters.

In this thesis, an intelligent controller based on a Raspberry Pi is presented. Proposed intelligent controller, which employs the PID (Proportional-Integral-Derivative) logic, is capable of controlling and maintaining the field of an electromagnet by adjusting the electrical current supplied to the coils of it, after the desired magnetic field is being entered by the end-user through a PC. The system comprises of a gaussmeter (with Hall sensor), power supply, electromagnet, PC and the intelligent controller based on a Raspberry Pi. The challenge of this project includes the integration of the intelligent controller into a standalone instrument, as a part of multi-component system.

For testing and evaluating the intelligent controller based on a Raspberry Pi, different manual and auto tuning methods are accommodated. The comparison of the shortlisted tuning configurations showed that a PI (Proportional-Integral) type of controller is the most applicable for this system. Results denoted that the prompted desire magnetic fields are successfully reached using the intelligent PI (Proportional-Integral) controller based on a Raspberry Pi. In terms of the system performance, magnetic field controlled process is stable, accurate and fast, without any signs of either overshooting or undershooting. Speed optimization is also undertaken, enhancing the response time of the system even more. Future work and directions are identified for further improvements in the system or in a variation of it.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/display.aspx?DocID=24420`), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library's regulations (see `https://www.library.manchester.ac.uk/about/regulations/_files/Library-regulations.pdf`) and in The University's policy on presentation of Dissertations.

Acknowledgements

This work is the final part of my postgraduate degree in Advanced Computer Science and IT Management at The University of Manchester, UK.

I would like to thank my supervisor Professor Thomas Thomson for his support and help throughout the project. His constructive feedback helped me improve my research writing, critical thinking, software problem-solving and experimental skills.

Finally, I would like to thank my friends and family for their words of encouragement and mental support throughout the project.

Giorgos Tsapparellas, PGT Student at The University of Manchester, 2019

The Author

Giorgos Tsapparellas received his BSc degree in Computer Science from Northumbria University, Newcastle, UK, in 2018, and awarded with ”Faraday Circuits Prize for Outstanding Achievement” for his distinct overall mark. After being awarded with Tom’s Kilburn scholarship, he is now furthering his studies with a MSc degree in Advanced Computer Science and IT Management at The University of Manchester, UK.

Previously, Giorgos served Cyprus’ military from the position of corporal (2012-14). During 2016 and 2017, he pursued a year-long industrial placement with Reece Innovation, in Newcastle, UK, from the position of Software Development Engineer. Then, during his final year of BSc studies, Giorgos has been recruited for six months as a part-time Consulting Software Engineer at Electrokinetic Limited, in Newcastle, UK.

Having always the desire to succeed, his research interests include Software Engineering, Automated Control Systems, Parallel Programming, Quantum Computing, Internet of Things (IoT), Business Analytics and IT Management. He is also a member of OR (Operational Research) society.

Chapter 1

INTRODUCTION

Considering technological advancements, recently, there has been growing interest in intelligent controllers which are widely used in industrial society. Straightforward applications of central heating and washing machine systems, as well as, more advanced applications of power steering, fuel pressure and liquid level control are being assisted by the intelligent controllers [1].

According to Bolton [1], automation describes the control of a process, including the execution of operations in the required sequence and controlling outputs to required values. A major benefit of automation is the little or no human intervention in the control process, a core requirement of the on-going Industry 4.0 revolution [1, 2, 3].

The first mechanical automatic feedback control system originally developed in England, UK, in the eighteenth century. In that period, an efficient steam engine governor introduced, where the device lie in the ”lift-tenter” mechanism solution which was used to control the gap between the grinding-stones in both water and wind mills [4].

Despite the usefulness of the automated mechanical applications, modern industrial society is constantly replacing mechanical methods utilized over the past decades with electronic solutions for automated control systems. This is due to the self-organizing, repeatable, reproducible, adaptive, robust and precise characteristics required by the modern control systems [5]. Electrical automated techniques are capable of providing these characteristics, improving the performance of control systems, and thus, increasing productivity and profitability of an organization [5].

Control system can be defined as the system which for some significant input and a set of desired goals is used to control a desired output [1]. According to Bolton [1], categories of control systems are controlling a variable to obtain the required value, controlling the sequence of events and controlling whether an event occurs or not.

For the proposed work, and considering the system requirements, controlling a variable to obtain the required value would be the most applicable category. In particular, project aims to introduce an intelligent controller based on a Raspberry Pi which will control the field of an electromagnet by adjusting the electrical current supplied to the coils of it, after the desired magnetic field is being entered by the user through a PC (Personal Computer). In a fast-paced manner, stable, as well as, accurate output should be produced by the intelligent controller, conforming with the prompted input. Intelligent controller based on a Raspberry Pi should also eliminate any possible overshooting in the system, ensuring that controlled output will never exceed the prompted input.

The following discussion includes the aim (see section 1.1.1), objectives (see section 1.1.2), deliverables and specification (see section 1.2), together with the outline (see section 1.3) of the project.

1.1 Aim and Objectives

1.1.1 Aim

The overall project aim is:

- Design, develop, test and evaluate an intelligent controller based on a Raspberry Pi that is used to control the magnetic field from an electromagnet.

1.1.2 Objectives

Having set a clear project aim/goal, more specific outcomes–objectives that would help achieve the desire aim/goal can be listed as follows:

- Design a support stand for Hall sensor and electromagnet.
- Assemble the required hardware components including the controller (Raspberry Pi), PC (Input and Output), power supply (with an IEEE 488.2 standard GPIB interface card), gaussmeter (including Hall sensor) and electromagnet.
- Design and develop the program for independent communication between the controller based on a Raspberry Pi and the PC to prompt the user to input a desired numerical magnetic field value and output the result at the end of the procedure.

- Design and develop the program for independent communication between the controller based on a Raspberry Pi and the gaussmeter (including Hall sensor) to compare the actual with the desired (user input) magnetic field value.
- Design and develop the program for independent communication between the controller based on a Raspberry Pi and the power supply (with an IEEE 488.2 standard GPIB interface card) to amend the electrical current going through the electromagnet, and thus, modifying the actual magnetic field as desired.
- Design and develop a mathematical model for the process that will be controlled.
- Integrate the communications of all instruments and implement the program that incorporates the PID (Proportional-Integral-Derivative) controller logic and uses tuning technique(s) to tune the controller based on a Raspberry Pi for maintaining the output stability, accuracy and speed (or settling time).
- Testing and evaluation of the usability, stability, accuracy, response and other performance measures of the assembled system.

1.2 Deliverables and Specification

Project is consisting of a single main deliverable which is the design, development, test and evaluation of an intelligent controller based on a Raspberry Pi. Ultimate goal would be the integration of the intelligent controller into a standalone instrument, as a part of multi-component system. Other hardware instruments of the system include PC, gaussmeter (with Hall sensor), power supply and electromagnet. Connections between the intelligent controller based on Raspberry Pi and other hardware instruments of the system should be established through widely-used and efficient Universal Serial Bus (USB) in conjunction with Ethernet and GPIB, if required.

After the desired magnetic field is being entered by the user through a PC, intelligent controller based on a Raspberry Pi should be able to control the field of an electromagnet by doing all the required calibrations and processing, including the adjustment of the electrical current supplied to the coils of the electromagnet. For a such closed-loop (with feedback signal) control system, stability, accuracy and speed (or settling time) characteristics should be considered. Intelligent controller based on a Raspberry Pi should also eliminate any possible overshooting in the system, ensuring that controlled magnetic field will never exceed the prompted magnetic field.

1.3 Outline

Sections 1.1 and 1.2 of this chapter explained the overall project aim together with the objectives that will help achieve this main goal and introduced the deliverables along with the specification of this project. The rest of this report is structured as follows:

- **Chapter 2 - Background and Theory:** This chapter critically reviews the literature related, but not limited to the control engineering, control systems, mathematical modelling of the control systems, process controllers and embedded systems.
- **Chapter 3 - Experimental Realization:** This chapter gives an overview of the mechanical, hardware and software elements of the project. Experimental realization includes also the evaluation of different microcontrollers for the intelligent controller and demonstrates how the system is assembled as a laboratory benchmark.
- **Chapter 4 - Design and Implementation:** This chapter discusses the design phase of the project, including system requirements specification, modelling of the proposed magnetic field control system, high-level software architecture and testing together with evaluation plans. Furthermore, implementation of the system is discussed, covering technical milestones of configurations, input method, gaussmeter control, power supply control, mathematical modelling of the process, PID (Proportional-Integral-Derivative) control technique, magnet calibration feature, UI, final software implementation and GUI.
- **Chapter 5 - Testing, Results, Evaluation and Optimization:** This chapter denotes the hardware and software unit tests that have been conducted, together with the integrated testing of the intelligent controller based on a Raspberry Pi. The integrated system's testing results are analyzed, discussed and evaluated, concluding to the fit-for-purpose tuning configuration for the intelligent controller. Speed optimization is also explored, enhancing the response time of the system.
- **Chapter 6 - Conclusions and Future Work:** This chapter summarizes the conclusions and achievements of the project, including a reflection on the set aim and objectives. Future work and directions are identified for further improvements in the system or in a variation of it.

Chapter 2

BACKGROUND AND THEORY

This chapter critically reviews the background and literature related to the control theory, including control engineering (see section 2.1), control systems (see section 2.2), mathematical modelling of the control systems (see section 2.3), process controllers (see section 2.4) and embedded systems (see section 2.5).

2.1 Control Engineering

Control engineering is a multi-disciplinary area with the interest of applying automated control theory in order to design and analyze optimum behavioural systems [5, 6]. According to Mandal [6], disciplines of control engineering include Mechanical Engineering, Computer Science and Electronics, or any other variation of these. The following discussion covers automation (see section 2.1.1) and control theory (see section 2.1.2).

2.1.1 Automation

Two of the most prolific contributors to the modern control systems, Dorf and Bishop [5] describe automation as the automatic procedure of controlling a device, process or a system as a whole. Automatic controllers have been shown a significant increase of productivity, enhancement of products quality, improvement of contingency plans, as well as, reduction of labor costs [5, 7], core requirements of the modern society infrastructure. Therefore, automation principle is being utilized over automobile, manufacturing and industrial segments, among others [5].

The history of automatic control can be broken down into four different periods, along with their main highlights as follows [4, 8]:

- **Early** Control period (until 1900): Main highlight - James Watt's (1769) steam engine and governor.
- **Pre-classical** Control period (1900-1940): Main highlight - Henry Ford's (1913) machine for automobile production.
- **Classical** Control period (1935-1960): Main highlight - George Devol's (1954) programmed article transfer (first industrial robot design).
- **Modern** Control period (after 1955): Main highlight - Introduction (1983) of the digital microprocessors.

Robots and driveless cars are some examples of automatic feedback control systems that are being experienced in our daily lives [5]. By doing a historical recursion on automatic control, it is clear that early mechanical methods have been replaced with electronic solutions due to modern industrial society demands.

Finally, yet importantly, although automation has been reportedly enhance countless markets, there are some disadvantages about it as well. High initial development costs and displacement of workers, due to little or no human intervention need are some concerns regards to automation [9, 10].

2.1.2 Control Theory

Control theory, a major contributor in technological, scientific and mathematical sectors since 1800s can be described as the way of continuously controlling dynamical processes or systems [11, 12]. Ultimate objectives of control theory include the control of a process or a system in satisfaction with stability, accuracy, speed (or settling time) and overshoot properties [13]. Discussion of SASO (Stability-Accuracy-Speed-Overshoot) properties and how they affect performance of control systems is followed in section 2.2.4.

According to Ogata [12], commonly used control theories include classical and modern control. As the basis of classical control foundations, root-locus and frequency-response techniques can form stable and satisfactory systems [12]. However, classical control is only applicable to single-input-single-output (SISO) systems. Therefore, for modern multi-variable systems (or industrial plants), where complexity is significantly

higher, modern control theory can be applied instead [12]. Modern control theory is a subsequent of digital computers (in the form of microcontrollers), which made viable the time-domain analysis and synthesis of complex systems [12].

In the modern industrial society, control theory is used, but not limited to robotics, home automation (e.g. central heating) and cruise control (e.g. car or aviation) systems [1, 5]. These modern automated control systems complying with high-performance, robustness, self-organization and adaption properties [5]. Therefore, by acquiring this sort of ”intelligence”, in the near future, modern automated control systems are expected to be fully operational with little or no human intervention [1].

2.2 Control Systems

Since early 1900s, automated control systems have been significantly enhanced engineering sector [4]. Some of the latest control system advancements include the development of intelligent micro-machines and functional nano-machines between 1988 and 2003 [8]. Control system can be described as the interconnection of elements, forming a system composition with ultimate goal of providing a desired outcome [5]. Figure 2.1 illustrates a block diagram of a process or a component that is being controlled in a system.

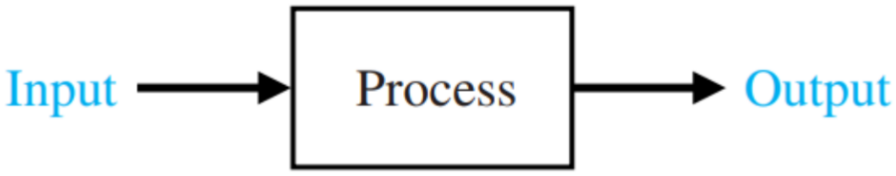


Figure 2.1: A block diagram of a process or a component that is being controlled in a system [5].

Designing a control system can be a complex procedure, especially when multiple variables or components have to be controlled. An optimal way of designing a control system is proposed by Dorf and Bishop [5], where design process comprises of seven different steps. Figure 2.2 shows the design procedure steps for a desired control system. If by the end of the design process, performance indices promptly meet the system requirements, design should be finalized. Otherwise, if the performance indices does not conform with the set of requirements, control system should be re-configured [5].

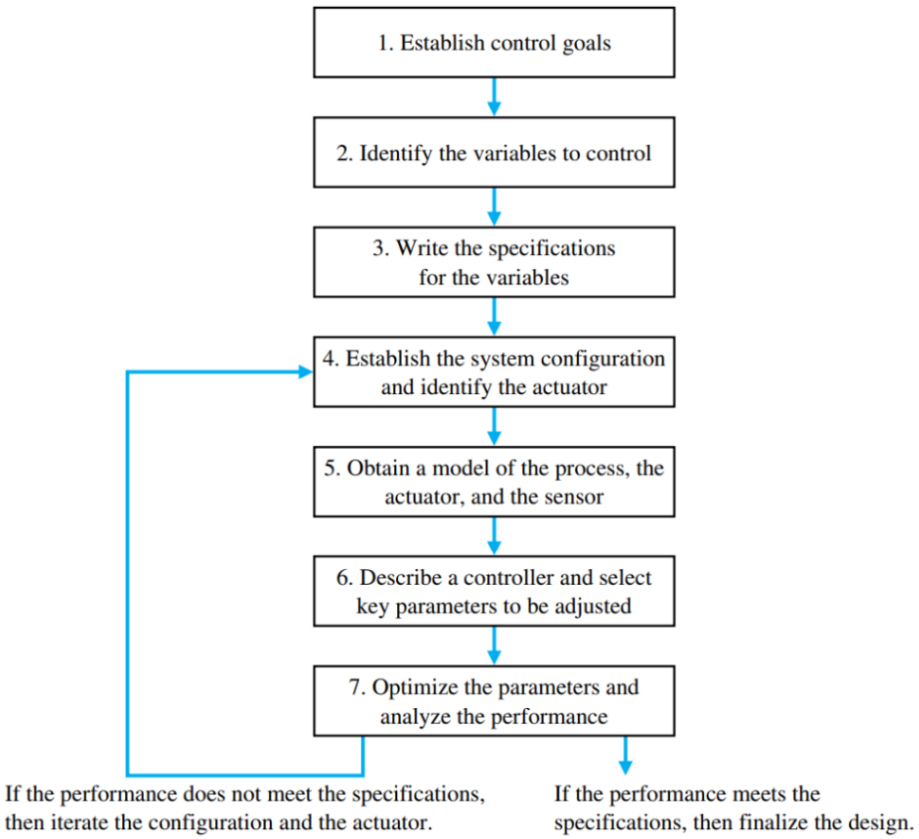


Figure 2.2: Design procedure steps for a desired control system [5].

According to Bolton [1], control systems can be categorized into three different variations, depending on their proposed functionality:

1. Control a variable to obtain the required value.

2. Control the sequence of events.

3. Control whether an event occurs.

Considering the first category, an example would be the control of room temperature using a central heating system [1]. For the second category, an example would be a belt used to feed blanks to a pressing machine, requiring a sequence of actions in order to be controlled [1]. Regards to the third category, an example would be the safety lock feature of a modern washing machine [1].

Extending above categorization, control systems can be further classified based on their application type. In particular, a control system can be characterized as either

digital, discrete-time or hierarchical [1]. Digital controllers are in the form of microcontrollers, providing essential accuracy and reducing possible noise in the system, while discrete-time controllers are controllers in which one or more input(s) can only change at discrete times in a specified period [1]. Hierarchical control refers to a control system used to control more systems, in a pyramid structure [1].

The following discussion includes open-loop control (see section 2.2.1), closed-loop control (see section 2.2.2), feedback control systems characteristics (see section 2.2.3) and SASO (Stability, Accuracy, Speed and Overshooting) properties (see section 2.2.4).

2.2.1 Open-Loop Control

Although feedback signal can be useful in some systems, there are numerous control systems where feedback signal is not required. Generally, a control system which uses an actuator or sensor to control a component or a process directly, without expecting feedback signal, can be described as open-loop [1, 5]. Therefore, as the procedure flow in the system follows only one direction, proposed output has no effect upon the signal to the process/component and it can be obtained by using either the input/set-value or its correction/modified value [5, 14, 15]. Figure 2.3 shows a block diagram comprising of the base elements of an open-loop control system (without feedback signal).

Figure 2.3: A block diagram comprising of the base elements of an open-loop control system (without feedback signal) [1].

An example of a simple open-loop control process would be a heating system for setting a room temperature to some required value [1]. Figure 2.4 shows the open-loop heating system controlling a room temperature with a switch feature.

Major disadvantage of an open-loop control system is that the process is being controlled based on the initial decision, and thus, no further modifications can be made [1]. Therefore, if for instance, output (in this case room temperature) is changed due to external factors (e.g. window opened), an open-loop system will not be able to fix it. In

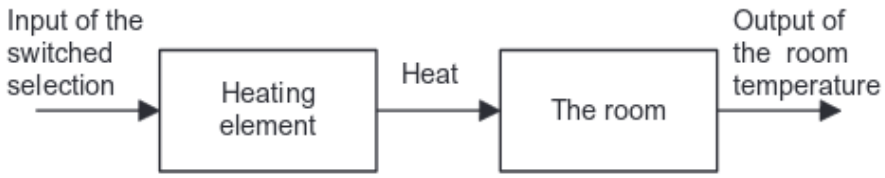


Figure 2.4: An open-loop heating system controlling a room temperature with a switch feature [1].

contrast, a similar closed-loop control system would be capable of eliminating such situation, using feedback signal feature [1].

2.2.2 Closed-Loop Control

Opposed to the open-loop control process, a closed-loop control system measures the output and uses feedback signal to compare it with the desired input (reference value or set-point) [1, 5, 14, 15]. Although the utilization of feedback signal enables the control of a desired output, and thus, it can improve accuracy of the system, it still demands a close consideration of stability and speed of response [5]. In the context of dynamic control, the most common closed-loop process controllers are PID (Proportional-Integral-Derivative) which will be discussed further in section 2.4.6. Figure 2.5 shows a block diagram comprising of the base elements of a closed-loop (with feedback signal) control system.

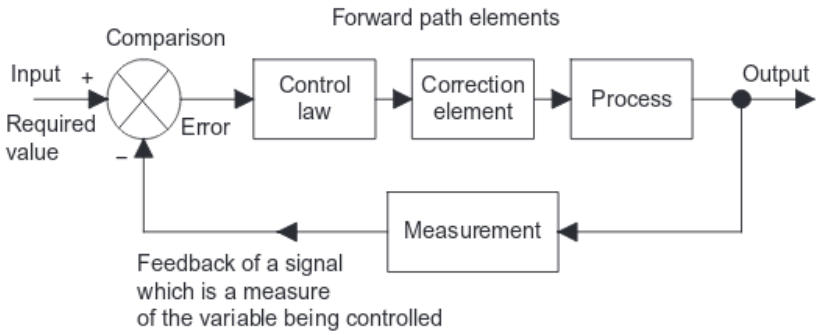


Figure 2.5: A block diagram comprising of the base elements of a closed-loop control system (with feedback signal) [1].

A more satisfactory example of a heating system, comparing to the simple open-loop control process (see section 2.2.1), would make use of feedback principle. Figure 2.6 shows the closed-loop electric heating system controlling a room temperature with a feedback signal feature.

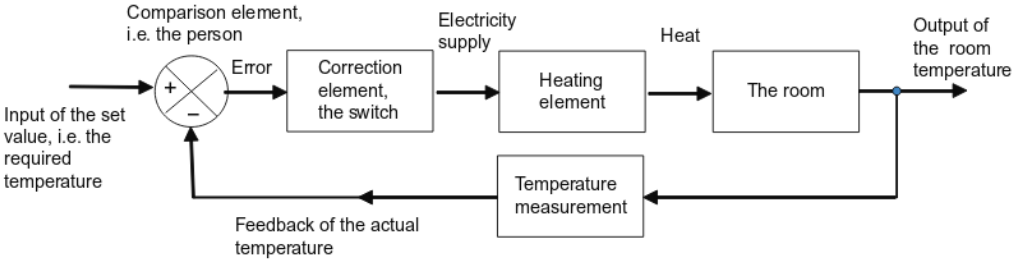


Figure 2.6: A closed-loop electric heating system controlling a room temperature with a feedback signal feature [1].

Major advantage of a closed-loop control system is that the process can be controlled and modified using a correction element, based on the feedback signal. In this case, feedback signal comprises of the constant comparison between required and actual temperatures [1]. Therefore, if for instance, output (in this case room temperature) is changed due to external factors (e.g. window opened), a closed-loop system will be able to fix it [1].

2.2.3 Feedback Control Systems Characteristics

Feedback principle has a vital role in the context of a closed-loop control system. Feedback can be defined as the process of either increasing or decreasing a manipulated variable (or feedback signal) when the measured variable is smaller or larger than the set-point (or reference input), respectively [1, 16]. Procedure of ”feeding back” a closed-loop control system is iterative with the ultimate goal of measured variable reaching the desired set-point (or reference input), using the manipulated variable as feedback signal [1]. Different types of feedback (or closed-loop) control systems exist which are listed as follows [17]:

- Single-Input-Single-Output (SISO)
- Single-Input-Multiple-Output (SIMO)
- Multiple-Input-Single-Output (MISO)

- Multiple-Input-Multiple-Output (MIMO)

According to Astrom and Murray [18], two of the most prolific contributors to the feedback control systems, the usage of feedback is often resulting in vast enhancements in control systems capabilities. This is due to feedback remarkable characteristics. Some of the key feedback characteristics (or properties) are robustness to uncertainty, dynamic design capabilities and higher-providence of automation [18].

While feedback utilization over closed-loop systems offers numerous advantages, it certainly has some disadvantages as well. This include drawbacks of instability (higher noise levels) and complexity of embedding a feedback controller in a multi-component system in the form of a microcontroller [18].

2.2.4 Stability, Accuracy, Speed and Overshoot (SASO) Properties

For a thorough measurement and optimization of control systems, stability, accuracy, speed and overshoot indices have to be considered closely. Stability, accuracy, speed and overshoot properties are also referred to as SASO [13].

Stability describes the ability of a control system to produce a desired output, while a bounded input is measured [1, 13]. In designing phase of control systems, stability is often considered as the top priority since unstable systems cannot provide the desired work at all [13]. Figure 2.7 shows a graphical representation of instability in a control system.

Furthermore, a control system is said to be accurate if the measured output is converged as close as possible to the set-point (or reference input), without undue oscillation [13]. Commonly, in order to find the accuracy level of a control system inaccuracy is measured (e.g. steady-state error) [13]. As an extension to accuracy property, repeatability is another useful performance indication for control systems which can be defined as the ability of resulting the same desired output for repeated measurements of the the same bounded input [1].

Moreover, speed (or settling time) is another important performance aspect for control systems. In most cases, control systems are set to be effective if they are responding in a fast-paced manner. In a control system, speed (or settling time) refers to the time that the desired output takes to settle with the set-point (or reference input) [13]. Figure 2.8 shows a graphical representation of both inaccuracy and speed (or settling time) in a control system.

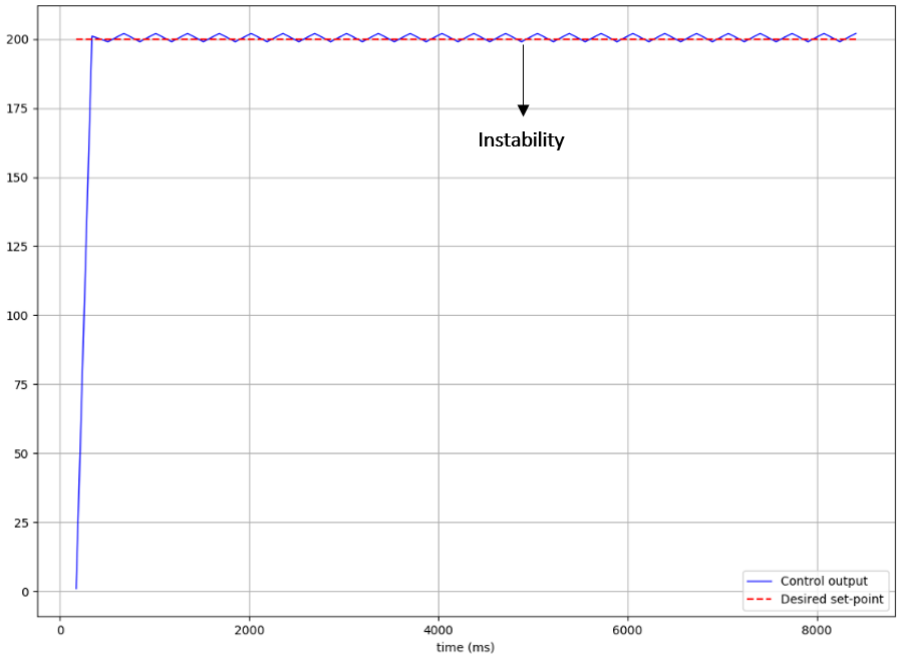


Figure 2.7: Graphical representation of instability in a control system.

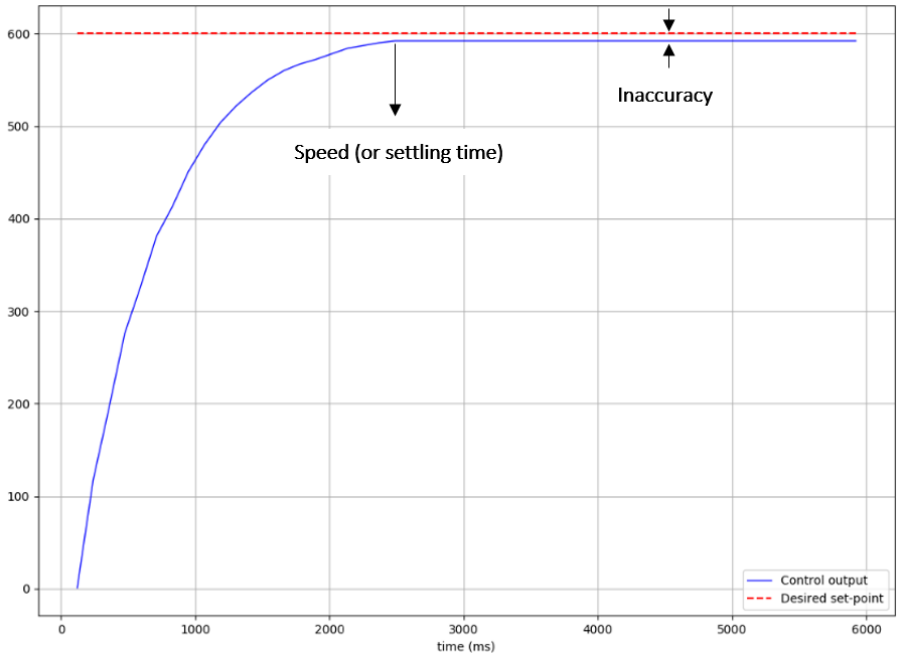


Figure 2.8: Graphical representation of inaccuracy and speed (or settling time) in a control system.

Additionally, overshoot property is another major aspect in control systems, especially when critical work is on-going. Overshoot can be described as the situation where measured output exceeds the set-point (or reference input) [13]. Figure 2.9 shows a graphical representation of overshooting in a control system.

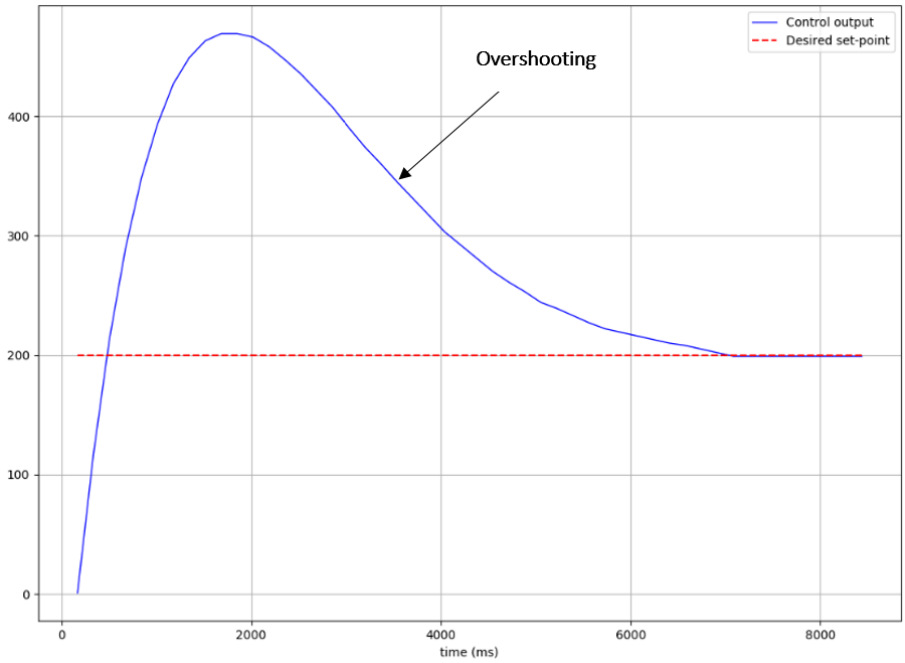


Figure 2.9: Graphical representation of overshooting in a control system.

In practice, meeting all four SASO (Stability-Accuracy-Speed-Overshoot) properties in a control system, simultaneously, can be challenging, but not impossible. This is due to ”trade-offs” that have to be considered at early stages of designing the control system [19]. For example, a common ”trade-off” case in a control system would involve the selection between stability and speed. If for instance, a control system responds quickly there is a high possibility that will also oscillates. Therefore, prioritization of SASO (Stability-Accuracy-Speed-Overshoot) properties is depending on the system’s requirements [19].

2.3 Mathematical Modelling of Control Systems

Mathematical modelling provides an easy way of representing a control system, as well as, giving an initial depth of how the process can be controlled, prior to moving to the design and implementation phases. The following discussion covers linear control (see section 2.3.1), non-linear control (see section 2.3.2) and transfer function (see section 2.3.3).

2.3.1 Linear Control

Closed-loop controllers can be categorized into linear and non-linear feedback controllers [15]. A linear control system generates an output as a continuous linear function of the input, where its dynamics adhere to the superposition property [15, 20].

A simple mathematical representation of a linear control system is shown through slope intercept equation 2.1.

$$y = m\,x + b, \tag{2.1}$$

where linear proportional relationship is defined between x and y as a result of m (the slope). Term b is referring to an optional offset to the output [20]. Indeed, equation 2.1 could mathematically represent proportional (P) control which will be discussed in section 2.4.1, along with other control modes as integral (I) (see section 2.4.2) and derivative (D) (see section 2.4.3).

Finally, yet importantly, despite the fact that linear modelling is utilized over various control systems, in reality, at some point, every system might outcome a non-linear behaviour under different circumstances [20].

2.3.2 Non-Linear Control

In contrast to linear control, non-linear control systems can be characterized as the ones where the input and the output do not exhibit a continuous linear relationship, and therefore, do not obey the superposition property [15, 20, 21]. For instance, in a non-linear control system the output may vary between two (or more) points as a function of a linear input level [20].

In mathematical form, non-linear control systems are often represented by non-linear differential equations as shown in equations 2.2, 2.3 and 2.4.

$$\frac{dy}{dx} = -y^2, \tag{2.2}$$

and

$$y = \frac{1}{x+Z}, \tag{2.3}$$

so

$$\frac{dy}{dx} + y^2 = 0, \tag{2.4}$$

where equation 2.3 is a general solution for the equation 2.2 and so it can be written in a non-linear form as in equation 2.4.

According to Mehrabi and McPhee [15], a non-linear control system can be linearized, through approximation, by using perturbation or Taylor series expansion techniques over a certain point. Consecutively, linear theory can be utilized for a non-linear model. However, linear model can only be effective if the non-linear model varies in a small range from the set point [15]. On the other hand, more optimal ways, specifically developed to handle non-linear control systems are limit cycle theory and multiple equilibria, ensuring sensible performance and stability of a non-linear model [15].

2.3.3 Transfer Function

Input, as well as, output are major contributors in a control system. In the context of an open-loop control system, input, and consecutively, output can be described as ”one-lap variables”, due to the fact that feedback signal is not acquired [1]. Therefore, once input and output are processed by the control unit, they can then discarded without the requirement of storing the previous input neither output in order to proceed to the next control iteration.

On the other hand, in the context of of a closed-loop control system, a re-formed input is repeatedly processed by the control unit in the form of feedback signal [1]. Above procedure is being executed until desired output is reached. Thus, in a closed-loop control system there is a clear relationship between input and output [1, 12].

Although input-output relationship of a closed-loop (or feedback) control system

is commonly demonstrated through differential equations, sometimes this technique might result into a complex process [1]. For instance, an example of a control system with two elements in series each having its input-output relationship described by differential equation can be considered [1]. In this case, an issue arises as it is difficult to observe how the output of the system as a whole is related to its input. One way of overcoming this problem is by using the Laplace transform [1].

The transfer function of linear time-invariant system can be described as the ratio of the Laplace transform of the output O(s) to the Laplace transform of the analogous input I(s), with all conditions assumed to be zero [12, 22]. According to Ogata [12], because of its simple algebraic form, transfer function is widely used for mathematical modelling of control systems. However, transfer function is only applicable for differential, time-invariant, linear systems [12].

A mathematical representation of transfer function in the context of a control system is shown in equation 2.5.

$$G(s) = \frac{O(s)}{I(s)}, \tag{2.5}$$

where G(s) is the transfer function as operator, O(s) is the output and I(s) is the input of a control system [1].

Alternatively, transfer function can also be illustrated as a block diagram (see Figure 2.10), where I(s) is the input and O(s) the output of a control system. Transfer function as operator can consecutively represented with G(s) symbol.

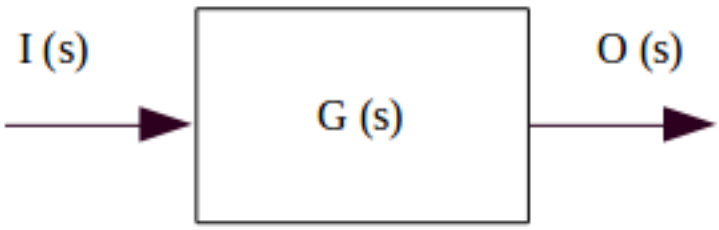


Figure 2.10: A block diagram illustrating the transfer function of a control system as G(s), input as I(s) and output as O(s) [1].

Extending above representation of transfer function, in a closed-loop (or feedback) control system there is an input of I(s), and thus, an output of F(s)O(s), in the form of feedback signal [1]. Error signal of such closed-loop control system is illustrated in

equation 2.6.

$$ErrorSignal(s) = I(s) - F(s)O(s), \tag{2.6}$$

where ErrorSignal(s) is the difference between the input I(s) and the feedback signal F(s)O(s) of the control system [1].

As long as, the feedback signal in a closed-loop control system can be either negative or positive, overall transfer function can then re-arranged to either equation 2.7 or equation 2.8 for negative and positive feedback, respectively.

$$Overall\ Transfer\ Function = \frac{O(s)}{I(s)} = \frac{G(s)}{1 + G(s)F(s)}, \tag{2.7}$$

where O(s) is the output of the system, I(s) is the input of the system, G(s) is the transfer function as operator and F(s) is the negative feedback [1].

$$Overall\ Transfer\ Function = \frac{O(s)}{I(s)} = \frac{G(s)}{1 - G(s)F(s)}, \tag{2.8}$$

where O(s) is the output of the system, I(s) is the input of the system, G(s) is the transfer function as operator and F(s) is the positive feedback [1].

Instead of transfer function, in a closed-loop (or feedback) control system, where there is no direct access to the input or output the alternative of manually conducting look-up tables can be used. For example, the value of an input/output (e.g. magnetic field) in a system might be a result of a controlled variable (e.g. electrical current supplied to the coils of an electromagnet). Therefore, correlation between the controlled variable and the input/output of such system can be obtained by matching all possible controlled variable-to-input/output correspondents. Practical example and discussion of look-up tables technique can be found in section 4.2.5.

2.4 Process Controllers

Process controllers can be described as control system components which are holding an input of the error signal, along with an output of a signal to modify the system output [1, 23]. According to Bolton [1], error signal refers to the difference between the required value and the measured actual value. Equation 2.9 represents a basic mathematical form of error signal in a control system.

$$e(t) = sp - pv, \tag{2.9}$$

where e(t) is the error signal in respect with time (t), sp is the set-point (or required value) and pv is the processed value (or measured value) [1].

Ultimate goal of a process controller would be to have zero error signal during the execution. However, due to modern control systems high-complexity and as the process control procedure is not necessarily limited to only one variable, indeed, avoidance of error signal in control systems can be challenging [23].

Control modes represent a common mechanism to acknowledge error signal changes in a control system [1]. For example, a simple process controller could be an ”on-off” device, where mode is set to ”off” when an error signal takes place [1]. An advanced example of a multi-variable process controller could be an engine processor, where controlled variables depending on the number of cylinders in the engine [23].

The following discussion includes proportional (P) control (see section 2.4.1), integral (I) control (see section 2.4.2), derivative (D) control (see section 2.4.3), proportional-integral (PI) control (see section 2.4.4), proportional-derivative (PD) control (see section 2.4.5), proportional-integral-derivative (PID) control (see section 2.4.6), proportional-integral-derivative (PID) tuning (see section 2.4.7) and proportional-integral-derivative (PID) limitations (see section 2.4.8).

2.4.1 Proportional (P) Control

Proportional (P) control, a type of linear feedback control system, can be characterized as the procedure where the size of controller output is proportional to the size of error signal by providing a control action through the ”all-pass” gain factor (Kp) [1, 24]. In mathematical form, proportional (P) control is represented through equation 2.10.

$$P = Kp\ e(t), \tag{2.10}$$

where P is the proportional term, Kp is the proportional gain and e(t) is the error signal with respect to time (t) [20, 24].

For a small error signal and when the input does not alter rapidly over time, proportional (P) control procedure is suggested [20, 25]. However, proportional (P) control cannot smoothly handle unforeseen events, and therefore, results to undershooting, overshooting and a hesitation to completely settle at the desired set-point [20]. For instance, if the proportional gain (Kp) is set too high, the system might result into oscillation. On the other hand, if the proportional gain (Kp) is set too low, the system will not be able to respond to input updates in a fast-paced manner [20].

Major drawback of proportional (P) control is that the process variable often deviates from the set-point (or reference input), known as steady-state error or ”droop” [1, 20, 25]. An example of Proportional (P) control steady-state error (or ”droop”) is shown in Figure 2.11.

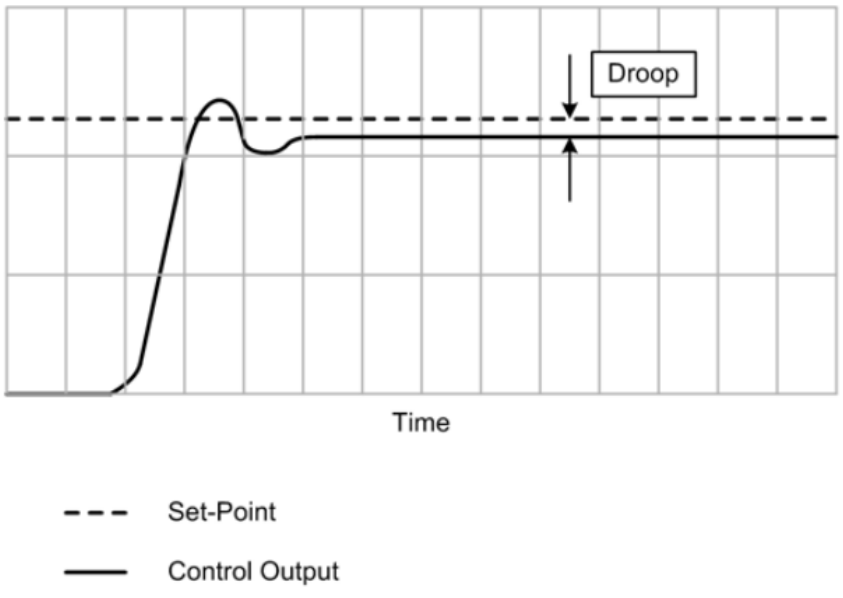


Figure 2.11: Proportional (P) control’s problematic characteristic of steady-state error (or ”droop”) [20].

2.4.2 Integral (I) Control

Integral (I) control can be defined as the mode where the controller output is proportional to the integral of the error signal with respect to time (τ) [1]. In mathematical form, equation 2.11 represents integral (I) control.

$$I = Ki \int_0^t e(\tau)\, d\tau, \tag{2.11}$$

where I is the integral term, Ki is the integral gain, e(τ) is the error signal with respect to integral interval time τ which may, or may not, be the same as t [20, 24].

For enhancing system performance, integral (I) mode is usually combined with proportional (P) mode (see section 2.4.4) or both proportional (P) and derivative (D) modes (see section 2.4.6).

2.4.3 Derivative (D) Control

Derivative (D) control can be defined as the action where the change in controller output from the set-point value is proportional to the rate of change with time of the error signal [1]. Equation 2.12 shows derivative (D) control in mathematical form.

$$D = Kd\, \frac{de(t)}{dt}, \tag{2.12}$$

where D is the derivative term, Kd is the derivative gain and e(t) is the error signal with respect to time (t) [20, 24].

According to Bolton [1], in a derivative (D) control procedure, controller output signal to the correction element given by derivative (D) controller is constant as the rate of change of the error signal with time is also constant.

The derivative (D) mode is not usually used alone, but in conjunction with proportional (P) mode (see section 2.4.5) or both proportional (P) and integral (I) modes (see section 2.4.6).

2.4.4 Proportional-Integral (PI) Control

One way to eliminate or mitigate steady-state error (or ”droop”) of proportional (P) mode is by combining both proportional (P) and integral (I) modes together [1, 20]. In

particular, integral (I) term, also known as the reset, has the capability of monitoring the gathered offset in the controller output and speed-up the controller output toward to the set-point (or reference input) [20].

For instance, as the controller output is the sum of the area starting from time t = 0, using the integral (I) term, even when the error has become zero the controller will be able to provide an output due to previous errors gathered [1].

In order to design a robust Proportional-Integral (PI) controller, proportional gain (Kp) should be lowered to account the inclusion of integral gain (Ki) [20]. It is important that if the integral gain (Ki) is set too high, overshooting and instability issues might occur [20]. Tuning methods (see section 2.4.7) can be used to find the effect of integral (I) term, and thus, the applicable integral gain (Ki) for a particular controller [20].

Equations 2.13, 2.14 and 2.15 show the Proportional-Integral (PI) control in different mathematical forms and for different observations.

$$PI = Kp\ e(t) + Ki \int_0^t e(\tau)\ d\tau, \tag{2.13}$$

where if the controller output becomes zero it can also referred to as

$$PI = Kp\ e(t) + \frac{Ki}{Kp} \int_0^t e(\tau)\ d\tau, \tag{2.14}$$

where Kp/Ki is called the integral action time (Ti) so

$$PI = Kp\ e(t) + \frac{1}{Ti} \int_0^t e(\tau)\ d\tau, \tag{2.15}$$

where PI is the Proportional-Integral controller output, Kp is the proportional gain, e(t) is the error signal with respect to time (t), Ki is the integral gain, e(τ) is the error signal with respect to integral interval time τ which may, or may not, be the same as t and Ti is the integral action time [1, 24, 25].

Commonly, Proportional-Integral (PI) control is preferred over a Proportional-only (P) control because of the absence of steady-state error (or ”droop”) [1, 20]. However, in order to prevent oscillations in the system and as the integration part is time-consuming, Proportional-Integral (PI) control might be slower than Proportional-only (P) control [1].

2.4.5 Proportional-Derivative (PD) Control

According to Bolton [1], Derivative-only (D) control can handle responses to changing error signals. However, since with a constant error the rate of change of error in respect to time is zero, Derivative-only (D) control cannot respond to constant error signals [1]. Therefore, in order to deal with constantly changing error signal derivative (D) control is combined with proportional (P) control [1].

In fact, using Proportional-Derivative (PD) control, an initial quick change occurs in the controller output due to derivative (D) action, followed by the gradual change because of the proportional (P) action [1]. Similarly to Proportional-Integral (PI) control, for designing a robust Proportional-Derivative (PD) controller, tuning (see section 2.4.7) should be accommodated. Therefore, effects of proportional (P) and derivative (D) terms can be observed, and thus, fit-for-purpose proportional (Kp) and derivative (Kd) gains can be selected for a desired controller.

In mathematical form, equation 2.16 shows the Proportional-Derivative (PD) control.

$$PD = Kp\, e(t) + Kd\, \frac{de(t)}{dt}, \tag{2.16}$$

where PD is the Proportional-Derivative controller output, Kp is the proportional gain, e(t) is the error signal with respect to time (t) and Kd is the derivative gain [1, 24].

Proportional-Derivative (PD) control is preferred over Proportional-only (P) control as it can deal with fast process changes in a better way [1]. However, in order to become fully operational, Proportional-Derivative (PD) control still requires a close consideration of steady-state error (or "droop") [1].

2.4.6 Proportional-Integral-Derivative (PID) Control

According to Astrom and Hagglund [26], in several cases, control systems are required to meet a complex set of design specifications. Therefore, Proportional-only (P), Proportional-Integral (PI) and Proportional-Derivative (PD) controls might not fit-the-purpose of advanced systems. Instead, Proportional-Integral-Derivative (PID), a more satisfactory control in terms of performance and robustness can be initiated by combining all three "control modes" [1, 27].

PID (or three-mode) controller is capable of precisely solving a wide-range of control issues by reducing the tendency for oscillations and eliminating the steady-state

error (or ”droop”) [1, 25]. Simple structure and opportunity of manual tuning are the major reasons of PID controllers success over various industrial applications [27].

In mathematical form, Proportional-Integral-Derivative (PID) algorithm is shown in equation 2.17.

$$PID = Kp\ e(t) + Ki \int_0^t e(\tau)\ d\tau + Kd\ \frac{de(t)}{dt}, \tag{2.17}$$

where PID is the Proportional-Integral-Derivative controller output, Kp is the proportional gain, Ki is the integral gain, Kd is the derivative gain, e(t) is the error signal with respect to time (t) and e(τ) is the error signal with respect to integral interval time τ which may, or may not, be the same as t [1, 16, 24].

In the context of a closed-loop (or feedback) control system, Figure 2.12 shows a block diagram of Proportional-Integral-Derivative (PID) control, where e is the error signal, u is the PID controller output, c is the system output and b is the feedback signal [20].

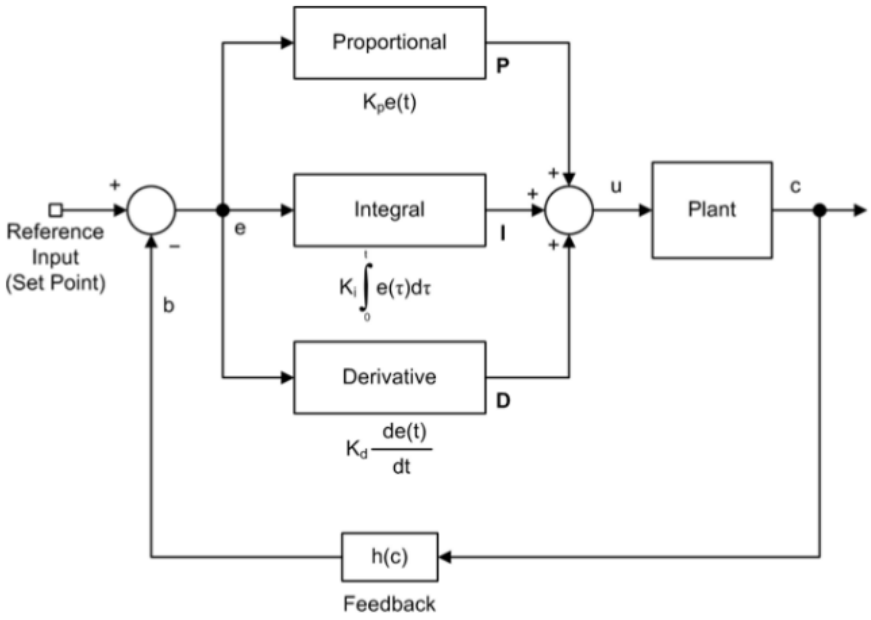


Figure 2.12: A block diagram of Proportional-Integral-Derivative (PID) feedback control system [20].

Although Proportional-Integral-Derivative (PID) control’s principle is used over numerous systems, in some cases, Proportional-Integral (PI) control is preferred, due to derivative (D) term behaviour. Generally, in a Proportional-Integral-Derivative (PID)

control system, the derivative (D) term acts to limit or prevent overshooting as the control output approaches the set-point (or reference input) [20]. However, at the same time, derivative (D) term slows the rate of change in the controller output, making the system slower [20]. On top of that, derivative (D) term tends to amplify noise, and thus, if the derivative gain (Kd) is not considered closely, control system might also become unstable [20]. More detailed discussion of the derivative (D) term's noise limitation in a Proportional-Integral-Derivative (PID) controller is followed in section 2.4.8.

2.4.7 Proportional-Integral-Derivative (PID) Tuning

According to Bolton [1], tuning can be described as the method that is being used to select the most optimum Kp, Ki and Kd gains for the proportional (P), integral (I) and derivative (D) modes, respectively. The selection of P, I and D control modes and their settings varies from system to system because of the different set of requirements [1]. PID tuning is directly affecting the performance of a control system, including stability, accuracy, speed (or settling time) and overshooting measures. Generally, PID tuning aims to mitigate or eliminate (if possible) some of the most common issues that can be confronted in a control system [1, 19]:

- Oscillations.
- Overshooting or Undershooting.
- Steady-state error (or "droop").
- High-frequency disturbance (or amplified noise).

The following discussion covers manual and auto tuning, including commonly-used methods of Ziegler-Nichols and Tyreus-Luyben [28, 29].

Manual Tuning

Manual tuning can be defined as the process of manually determining proportional (Kp), integral (Ki) and derivative (Kd) gains in a closed-loop (or feedback) control system. Individual effects of manually tuning the proportional (Kp), integral (Ki) and derivative (Kd) gains are [20]:

- Proportional gain (**Kp**) is controlling the rise time in a PID control system. Increasing Kp gain in a PID control system would result into faster rise time, but

with the possibility of overshooting and reduced speed (or longer settling time). On the other hand, decreasing the Kp gain in a PID control system would result into slower rise time, but with less (or no) signs of overshooting. The utilization of Kp gain by itself, without the inclusion of Ki and Kd gains would result steady-state error (or ”droop”).

- Integral gain (**Ki**) is eliminating the steady-state error (or ”droop”) in a PID control system. However, if the Ki gain is set too high, speed of the PID control system may be reduced and controlled output might overshoot. Therefore, conjunction of both Kp and Ki gains must be considered closely in order to achieve optimal rise time, together with minimal overshooting in a PID control system.
- Derivative gain (**Kd**), if tuned properly, it can add value to a PID control system by reducing overshoot, improving stability and increasing the speed (or reducing the settling time). However, if the Kd gain is set too high, PID control system may become unstable and go into oscillation because of high-frequency disturbance (or amplified noise).

Table 2.1 summarizes the performance effects of manually increasing each proportional (Kp), integral (Ki) and derivative (Kd) gains in a PID control system.

Closed-Loop Response	**Rise Time**	**Overshoot**	**Settling Time**	**Steady-State Error**	**Stability**
Increasing Kp	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing Ki	Small Decrease	Increase	Increase	Large Decrease	Degrade
Increasing Kp	Small Decrease	Decrease	Decrease	Minor Change	Improve

Table 2.1: Performance effects of manually increasing each proportional (Kp), integral (Ki) and derivative (Kd) gains in a PID control system [24].

Manually tuning the proportional (Kp), integral (Ki) and derivative (Kd) gains of P, I and D control modes provides flexibility in the control procedure, and thus, better performance results [1, 24]. However, conducting manual tuning for the purposes of a PID controller can be time-consuming and requires experience. Therefore, auto tuning methods of Ziegler-Nichols (or ultimate cycle) and Tyreus-Luyben can be used instead

to tune a PID controller [28, 29].

Auto Tuning

Manual tuning cannot be automated as it is based on trial and error procedure. On the other hand, Ziegler-Nichols (or ultimate cycle) and Tyreus-Luyben methods can be used to auto tune a PID controller [1, 28, 29].

In order to apply Ziegler-Nichols auto tuning method in a closed-loop (or feedback) control system, certain steps have to be performed [1]:

1. Set the integral (Ki) and derivative (Kd) gains to zero.
2. Set the proportional gain (Kp) to a low value and introduce a set-point (or reference input).
3. Increase the proportional gain (Kp) until the system goes to sustained periodic oscillation.
4. Record the proportional gain (Kp) caused the system to go into sustained periodic oscillation and measure the oscillation period (Tu).
5. These values are referred to as the ultimate gain (Kpu) and the ultimate oscillation period (Tu).

The extracted ultimate gain (Kpu) and ultimate oscillation period (Tu) values can then be used to calculate the proportional (Kp), integral (Ki) and derivative (Kd) gains, using Ziegler-Nichols recommended criteria of Table 2.2.

Type of controller	**Kp**	**Ti**	**Td**
P-only	0.5Kpu	-	-
PI	0.45Kpu	Tu/1.2	-
PID	0.6Kpu	Tu/2	Tu/8

Table 2.2: Ziegler-Nichols recommended criteria for auto tuning a P-only, PI (Proportional-Integral) and PID (Proportional-Integral-Derivative) type of controllers [1].

In order to conduct Tyreus-Luyben auto tuning method in a closed-loop (or feedback) control system, the same steps with Ziegler-Nichols method have to be followed [30]. However, in order to calculate the proportional (Kp), integral (Ki) and

derivative (Kd) gains, the recorded ultimate gain (Kpu) and ultimate oscillation period (Tu) values have to be applied to the Tyreus-Luyben's recommended criteria shown in Table 2.3. It is important that Tyreus-Luyben auto tuning method does not include the P-only control mode.

Type of controller	Kp	Ti	Td
PI	Kpu/3.2	2.2Tu	-
PID	Kpu/2.2	2.2Tu	Tu/6.3

Table 2.3: Tyreus-Luyben recommended criteria for auto tuning a PI (Proportional-Integral) and PID (Proportional-Integral-Derivative) type of controllers [30].

Since for both Ziegler-Nichols and Tyreus-Luyben auto tuning methods, ultimate gain (Kpu) is equals to the proportional gain (Kp) and Ti together with Td can be calculated from the tables above, integral (Ki) and derivative (Kd) gains can be calculated using the following equations [1]:

$$Ki = Kp/Ti, \tag{2.18}$$

and

$$Kd = Kp * Td, \tag{2.19}$$

2.4.8 Proportional-Integral-Derivative (PID) Limitations

Proportional-Integral-Derivative (PID) algorithms are assisting more than 90% of industrial the controllers because of their simplicity and applicability [24]. However, there are some limitations regarding standard Proportional-Integral-Derivative (PID) controllers which are concerning industrial and academic research for a while now.

In a Proportional-Integral-Derivative (PID) controller, if tuned properly, derivative (D) term can add value into the process and maximize system performance by providing useful phase load to offset phase lag introduced by the integral (I) term [24, 31]. Nevertheless, according to Astrom and Hagglund [16], derivative (Kd) gain can cause a high-frequency measurement noise in a control system which leads to larger amplitude

of the controlled output. To prevent this, the majority of active Proportional-Integral-Derivative (PID) controllers do not make use of derivative (D) term, limiting fully exploitation of the controller features [24].

Furthermore, linearity of Proportional-Integral-Derivative (PID) controllers is limiting any non-linear effects that must be counted in a control system [16]. For instance, in the context of a closed-loop (or feedback) control system, if there is a large change of the set-point (or reference input) the integral (I) term may become very large, causing overshooting and instability [16, 24, 31]. This phenomenon is also called ”integrator windup” [16]. A common way to avoid ”integrator windup” in a control system is by introducing ”anti-windup” in the form of pre-determined set-point bounds [16]. However, this approach might limit Proportional-Integral-Derivative (PID) controller performance and it does not guarantee the avoidance of ”windup” caused by disturbances [16].

In order to improve Proportional-Integral-Derivative (PID) controllers functionality and performance, different modifications have been conducted to the standard Proportional-Integral-Derivative (PID) algorithm [16]. Some modifications include the introduction of Proportional-Integral-Derivative (PID) error-squared (or gain scheduling) [16], the addition of feed-forward principle [32] and the implementation of the fuzzy logic [33].

2.5 Embedded Systems

According to Pont [34], embedded system refer to an application or a programmable computer (in the form of a microcontroller) which is developed by humans in order to handle a set of tasks or control a process. Due to their dynamic and fast-paced providence, embedded systems are widely used over the modern technological society [1, 34]. Depending on its functionality, an embedded system can be characterized as either event-triggered or time-triggered [34].

In an event-triggered (or event-driven) embedded system, a set of tasks or activities are initiated after the occurrence of a specific event [35]. For instance, a desktop application should handle mouse movements or clicks in an event-triggered manner [34]. Another example of an event-driven embedded system would be in the form of a feedback control system, where a parameter (e.g. supplied electrical current) is set to change after a sensor reading (e.g. Hall sensor) to control an output (e.g. magnetic field).

On the other hand, in a time-triggered embedded system, a set of tasks or activities are handled periodically, based on a pre-set scheduling [35]. For example, a time-triggered system could be a microwave oven, where the state (on/off) of the system is based on a set timer [34]. Another example of a safety-critical time-triggered system would be the control of road traffic lights.

Although there is a long debate on which embedded systems perform better, event-triggered or time-triggered, there is no clear answer. It is purely depending on system requirements [36]. For instance, due to their strictly timed configuration, time-triggered systems are more predictable, and thus, perform better under safety-critical systems [36].

Conclusively, in order to successfully design and develop an ”intelligent” embedded control system, either event-triggered or time-triggered, analysis and selection of the microcontroller must be considered closely.

Chapter 3

EXPERIMENTAL REALIZATION: MECHANICAL, HARDWARE AND SOFTWARE ELEMENTS

In this chapter, we will explore and specify the mechanical (see section 3.1), hardware (see section 3.2) and software (see section 3.4) elements of the project. Evaluation of different microcontrollers for the intelligent controller is also included, together with a demonstration of assembled closed-loop (or feedback) control system, as a laboratory benchmark (see section 3.3).

3.1 Mechanical Instrumentation

3.1.1 Probe Stand

The Lake Shore probe (or Hall sensor) can be flagged as the most sensitive instrument of the control system. Clamping to the probe stem can alter probe's calibration and in excessive situations may also completely destroy the Hall sensor [37]. Therefore, it is highly-recommended that the probe is being held in place by securing it at the handle. Lake Shore Cryotronics does offer two different "off-the-self" probe stands for mounting the Hall sensor. However, Lake Shore's probe stands with version numbers *4030-12* and *4030-24* are priced at $\sim$\$300 [38]. Considering the cost-effectiveness and in order to ensure that the probe is held exactly in place for accurate field measurements, an in-house aluminium probe stand was designed by the author and manufactured in the University's workshop by the employees.

The probe stand constitutes of a base with two M6 drilled and tapped holes for mounting the probe holders, along with a secured place for adjusting the electromagnet. Two different units (bottom and top) with a semi-hole on each are acting as the probe holders, responsible for holding the Hall sensor in place (centered between the magnets of the electromagnet). Figure 3.1 shows the probe stand from three different angles, together with a representation of both the probe (or Hall sensor) and the electromagnet being mounted on it. Design schematics of the in-house designed aluminium probe stand (including the base) can be found in Appendix A-Figure A.1.

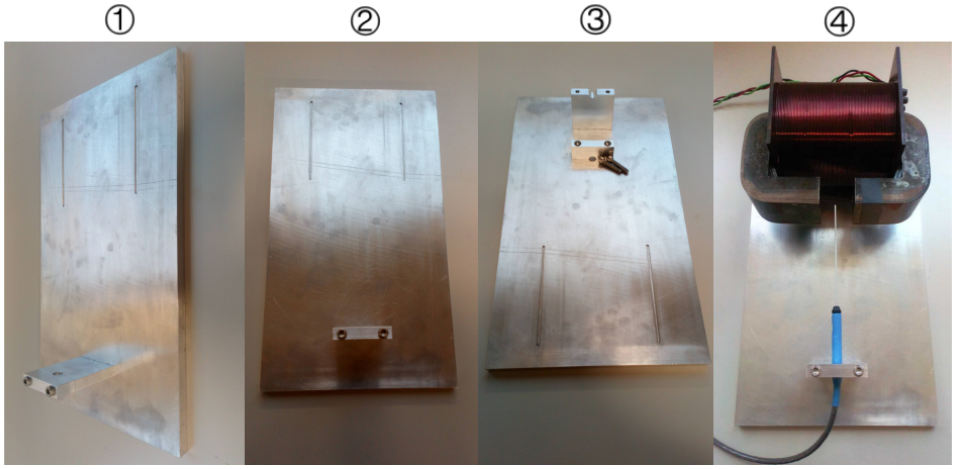


Figure 3.1: In-house designed probe stand's side view is marked with 1, top view with 2 and front view with 3. Representation of both the probe and the electromagnet being mounted on the stand and the base, accordingly, is marked with 4.

3.2 Hardware Instrumentation

The following discussion will give a brief overview of the hardware elements of the system, including Lake Shore 425 gaussmeter (see section 3.2.1), Kepco BOP power supply (see section 3.2.2), electromagnet (see section 3.2.3) and intelligent controller microcontroller (see section 3.2.4). Full list of hardware instruments can be found in Appendix B-Table B.1.

3.2.1 Lake Shore 425 Gaussmeter

Lake Shore 425 gaussmeter along with its high sensitivity probe (HSE), manufactured by Lake Shore Cryotronics, are responsible for reading the magnetic field values in

the system. Main features of Lake Shore 425 gaussmeter are shown in Table 3.1. Figure 3.2 shows Lake Shore 425 gaussmeter along with its probe (or hall sensor). Lake Shore 425 gaussmeter is giving the opportunity of choosing a field unit between Gauss, Tesla, Oersted and Ampere/meter selections. For this project, Oersted (Oe) magnetic field unit has been chosen.

Feature	Details
Liquid Crystal Display (LCD)	Displaying feedback of the operation (field value, unit etc.)
RS-232 serial port	Fixed 57,600 baud rate for Universal Serial Bus (USB) interface
Probe (or hall sensor)	High sensitivity probe (HSE) for reading the magnetic field values
Alarm with relay	-
Reading rate	30 readings per second
Magnetic field units	1 - Gauss 2 - Tesla 3 - Oersted 4 - Ampere/meter
DC to 10KHz AC frequency	-
DC accuracy	+-0.20%
Magnetic field range	350mG-350kG

Table 3.1: Main features of Lake Shore 425 gaussmeter (information from [37]).

(a) Lake Shore 425 gaussmeter.

(b) High sensitivity probe (HSE).

Figure 3.2: Lake Shore 425 gaussmeter along with its probe (or hall sensor) [37].

3.2.2 Kepco BOP Power Supply

Kepco BOP power supply and National Instruments GPIB-USB-HS control device are manufactured by Kepco and National Instruments, accordingly. In the scope of this project, Kepco BOP power supply and National Instruments GPIB-USB-HS control device are used to remotely adjust the electrical current supplied to the coils of the electromagnet. Main features of Kepco BOP power supply are shown in Table 3.2.

Feature	Details
Front Panel Display	Analog
Supply power	100/200/400W (Watts)
Supply voltage	+-20V (Volts)
Supply current	+-20A (Amps)
Limits capability	For supply voltage and current
BIT 4886 digital interface card	Compatible with NI GPIB-USB-HS control device
Remote control	Through GPIB-USB and SCPI commands

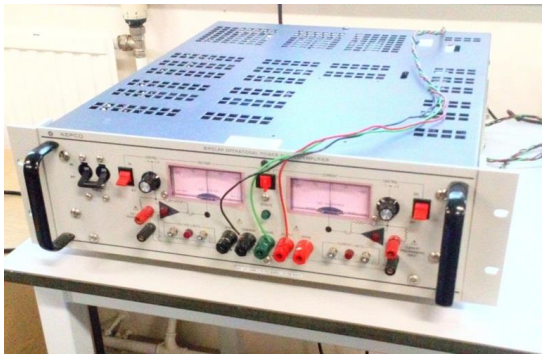
Table 3.2: Main features of Kepco BOP power supply (information from [39, 40]).

It is important that both physical and remote controlling of Kepco BOP power supply are not applicable at the same time. Kepco's BOP power supply voltage and current values can be controlled either remotely by an interface program (e.g. intelligent controller) or physically using the front panel [39, 40]. Figure 3.3 illustrates Kepco BOP power supply and National Instruments GPIB-USB-HS control device.

National Instruments GPIB-USB-HS control device

For the proposed magnetic field control system, GPIB-USB-HS control device is responsible for bridging the Kepco BOP power supply with the Raspberry Pi (controller). General Purpose Interface Bus (GPIB) wire-end is connected to the Kepco's BOP power supply socket, while Universal Serial Bus (USB) wire-end is plugged to the Raspberry's Pi (controller) associated port.

Remote communication between the Kepco BOP power supply and the controller (Raspberry Pi) can then established by using the Standard Commands for Programmable Instruments (SCPI), offered by BIT 4886 digital interface card [40]. Main specifications of National Instruments GPIB-USB-HS control device are shown in Table 3.3.



(a) Kepco BOP power supply.

(b) NI GPIB-USB-HS control device [41].

Figure 3.3: Kepco BOP power supply and National Instruments GPIB-USB-HS control device.

Specification	Details
Performance	Up to 7230 kbytes/s
Power (USB bus)	+5V (Volts)
Dimensions	10.7cm x 6.6cm x 2.6cm
Weight	180g
GPIB connector	IEEE 488 24-pin connector
USB connector	USB standard Type A plug
Operating temperature	0 °C to 55 °C

Table 3.3: Main specifications of National Instruments GPIB-USB-HS control device (information from [42]).

3.2.3 Electromagnet

In order to create the desire magnetic field for the proposed feedback control system, an electromagnet is used, powered by Kepco BOP power supply. Generally, electromagnets are in the form of iron cored solenoids, where ferromagnetic property of iron core forces the magnetic domains of the iron to align with the smaller driving magnetic field produced by the current in the solenoid [43]. Electromagnet of the proposed work has 1.18mm wire diameter and consists of 1614 turns. For safety purposes and due to relatively small coils of the proposed electromagnet, supplied electrical current should not exceed +-3 Amps. Figure 3.4 represents the electromagnet of the system.

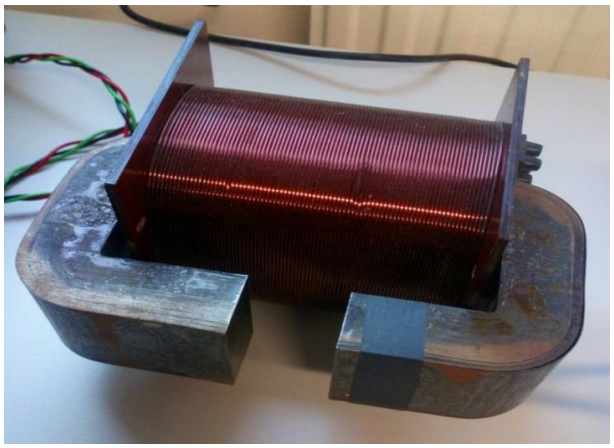


Figure 3.4: Electromagnet of the proposed magnetic field control system.

3.2.4 Intelligent Controller Hardware Specification

After defining a set of requirements and evaluating the shortlisted microcontrollers, a fit-for-purpose intelligent controller has been chosen. The following discussion includes microcontroller set of requirements, Arduino Mega 2560 R3, Raspberry Pi 3 Model B+, Odroid-XU4 and microcontrollers evaluation.

Microcontroller Set of Requirements

Considering the project aim, time-frame, budget and other hardware components of the system (see sections 3.2.1 and 3.2.2), following set of requirements have been identified for the selection of intelligent controller hardware:

1. A minimum of 2 Universal Serial Bus (USB) ports for sufficient interface with the other hardware instruments of the system (power supply and gaussmeter).

2. Familiarity with the programming language offered in accordance to the microcontroller.

3. Cost-effectiveness.

4. Reasonable performance, power-consumption and storage.

5. Portability.

Arduino Mega 2560 R3

Leonardo, Uno and Mega are some of the hardware devices Arduino is offering. However, Arduino Mega 2560 R3 microcontroller is designed for more advanced projects similar to the proposed work. Arduino Mega 2560 R3 board has 256KB (flash) memory, 1 Universal Serial Bus (USB) port, 54 digital I/O pins, 16 analog inputs, a single core AT mega 2560 microcontroller and can be programmed over Arduino IDE (C/C++) [44]. Figure 3.5 illustrates the Arduino Mega 2560 R3 board.

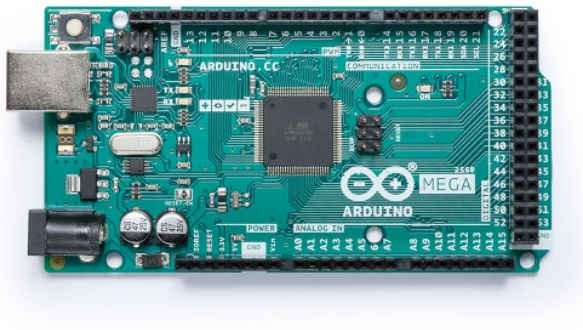


Figure 3.5: Arduino Mega 2560 R3 board [44]. A hardware option for intelligent controller.

Raspberry Pi 3 Model B+

Raspberry is offering various devices as Pi Model A+, Pi 2 Model B and Pi 3 Model B+. Although all Raspberry Pi board versions have similar specifications, latest release of Raspberry Pi 3 Model B+ offers improved functionality and speed [45, 46]. As a compact computer board, Raspberry Pi Model 3 B+ includes its own Linux-based (NOOBS or Raspbian) operating system (OS) along with ARM Cortex-A53 processor, 4 Universal Serial Bus (USB) ports, 4 number of cores and 1GB of memory. External micro SD card for loading the operating system (OS) and storing data is also available [47, 48]. In terms of programming the Raspberry Pi Model 3 B+, Python (recommended) and C/C++ languages are applicable [49]. Figure 3.6 shows the Raspberry Pi Model 3 B+ computer board.

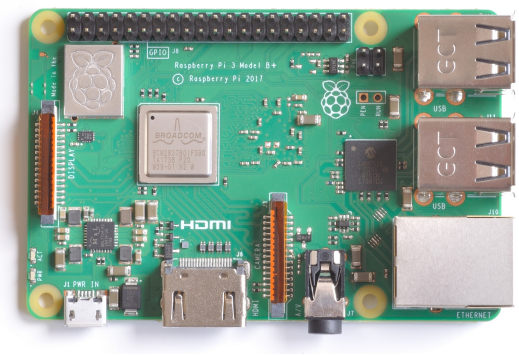


Figure 3.6: Raspberry Pi Model 3 B+ computer board [47]. A hardware option for the intelligent controller.

Odroid-XU4

N2, C2 and XU4 are the different board versions offered by Odroid. Odroid-XU4 release provides the most energy-efficient and powerful solution [50]. Furthermore, as a computer board, Odroid-XU4 is powered by ARM big.LITTLE technology and includes its own Linux-based (Ubuntu or Android) operating system (OS). Four Universal Serial Bus (USB) ports, 8 number of cores and 2GB of memory are also available with Odroid-XU4. Both, Python and C/C++ languages can be used to program Odroid-XU4 computer board [50]. Figure 3.7 presents the Odroid-XU4 computer board.

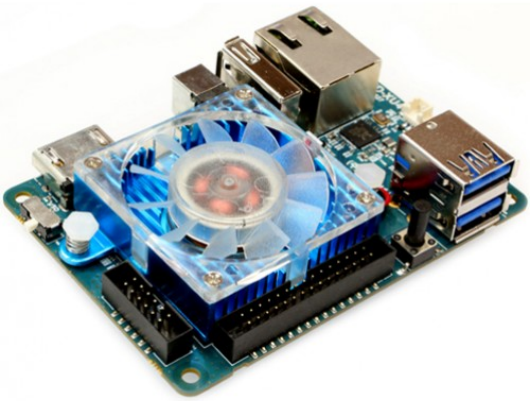


Figure 3.7: Odroid-XU4 computer board [51]. A hardware option for the intelligent controller.

Microcontrollers Evaluation

Considering the identified set of microcontrollers requirements (see section 3.2.4), comparison of Arduino Mega 2560 R3, Raspberry Pi 3 Model B+ and Odroid-XU4 microcontrollers is shown in Table 3.4.

	Arduino Mega 2560 R3	**Raspberry Pi 3 Model B+**	**Odroid-XU4**
Microcontroller /Processor	AT mega2560	ARM Cortex-A53	ARM big.LITTLE (Heterogeneous)-4xARM Cortex-A15 and 4xARM Cortex-A7
Input/Voltage	7-12V	5.1V	5V
CPU/Clock Speed	16MHz	1.4GHz	2GHz
Memory	256kB (Flash)	1GB	2GB
USB ports	1	4	3
No. of Cores	Single	Quad	Octa
OS	Linux, Windows and macOS	Linux (NOOBS or Raspbian)	Linux (Ubuntu or Android)
Programming Language	Arduino IDE (C/C++)	Python (recommended) and C/C++	Python and C/C++
Length	101.52mm	85mm	82mm
Width	53.3mm	49mm	58mm
Weight	37g	42g	60g
Price[1]	£30.00	£35.00	£60.00

Table 3.4: Comparison between Arduino 2560 R3, Raspberry Pi 3 Model B+ and Odroid-XU4 hardware options for the selection of the intelligent controller (information from [44, 45, 46, 47, 48, 49, 50, 51]).

Although the most cost-effective hardware choice and familiar programming environment, Arduino Mega 2560 R3 microcontroller was not selected as the desired intelligent controller. This is mainly due to single Universal Serial Bus (USB) port

[1]Prices in accordance to February 2019.

support. Therefore, additional Universal Serial Bus (USB) shield would be required or further implementation of Universal Serial Bus (USB) communication on software level which would be time-consuming, considering the available time-frame of the project.

For Odroid-XU4 computer board, despite the fact that is the most energy-efficient and powerful (in theory) hardware option, it cannot provide something more than Raspberry Pi at this stage, at a higher cost. It is fundamentally designed for parallel computing which is out of the project scope.

Considering the above technical evaluation and any other ”trade-offs” between the three microcontrollers, Raspberry Pi 3 Model B+ selected as the most suitable hardware component for the intelligent controller. Raspberry Pi covers all five set microcontroller requirements (see section 3.2.4).

3.3 Assembling the Control System

The project aims to control the field of an electromagnet by adjusting the electrical current supplied to the coils of it, after the desired magnetic field is being entered by the user through a PC (Personal Computer). System comprises of PC (Input and Output), Raspberry Pi (controller), Lake Shore 425 gaussmeter (including Hall sensor), electromagnet and Kepco BOP power supply (including an IEEE 488.2 standard GPIB interface card). Figure 3.8 shows the assembled closed-loop (or feedback) control system, as a laboratory benchmark.

End-to-end wired connections between the intelligent controller based on a Raspberry Pi and the other hardware instruments of the system have been established through combinations of widely-used and efficient Universal Serial Bus (USB), General Purpose Interface Bus (GPIB) and Ethernet, where applicable. Intelligent controller based on a Raspberry Pi, PC, Lake Shore 425 gaussmeter and Kepco BOP power supply are powered by their dedicated cables, while the electromagnet is powered using the phase (red), neutral (black) and protective ground (green) wires.

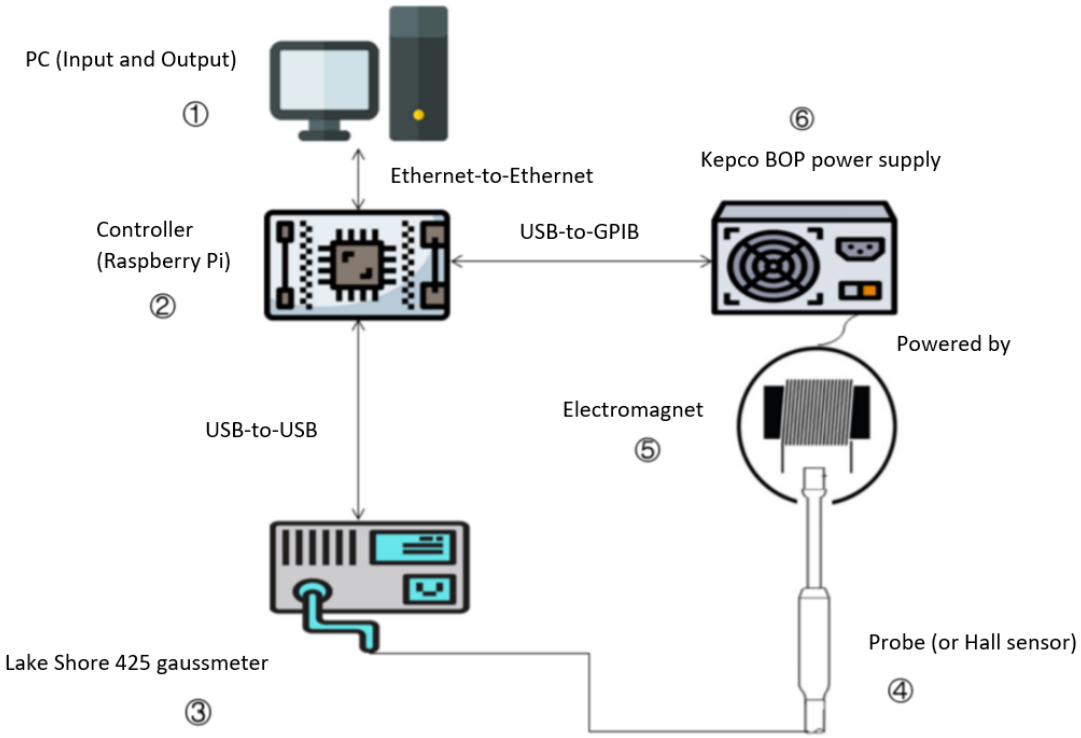


Figure 3.8: Assembled closed-loop (or feedback) control system overview. PC is marked with 1, controller (Raspberry Pi) with 2, Lake Shore 425 gaussmeter with 3, Probe (or Hall sensor) with 4, electromagnet with 5 and Kepco BOP power supply (including an IEEE 488.2 standard GPIB interface card) with 6.

3.4 Software and Tools

The following discussion will give a brief overview of the software and tools of the system. Full list of the software and tools can be found in Appendix C-Table C.1.

3.4.1 Python Software Language

A high-level software language, Python, originally developed in the late 1980s by Guido van Rossum [49]. Due to its object-oriented providence, flexibility and clear syntax, Python is continuously gaining momentum in engineering, mathematics, computing and physics sectors. Moreover, Python is an open-source software language, freely available for OS X, Windows and Linux environments [49].

Although Python source code can be produced over IDLE Shell, for this project, Thonny IDE is used to initiate and maintain the software framework. Thonny IDE

is a simple and easy-to-use Python environment, coming pre-installed with Raspberry Pi [52]. Figure 3.9 shows the Thonny IDE. For compiling and running produced software, Python version 2.7 is used.

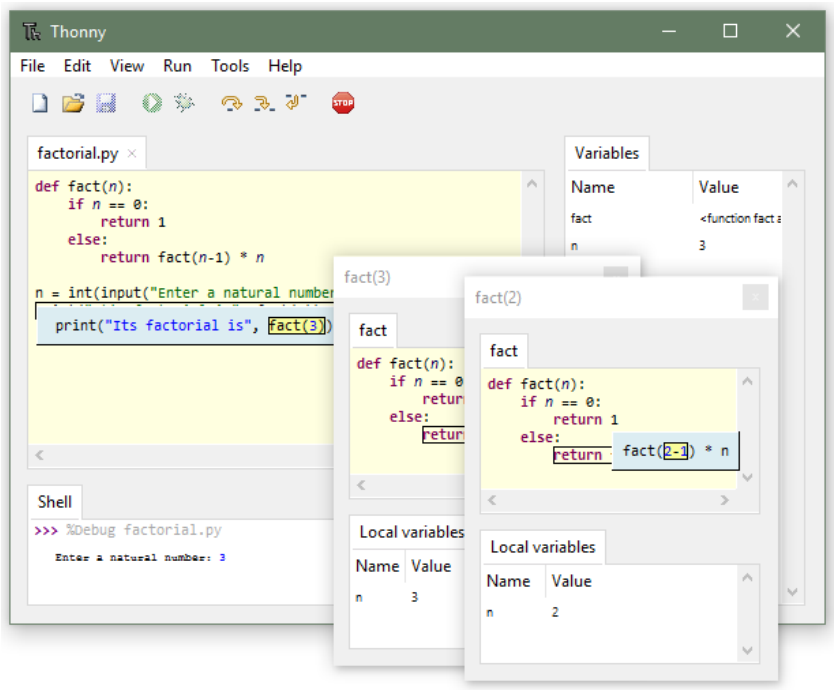


Figure 3.9: Thonny Python IDE [53].

Finally yet importantly, Python was chosen due to its applicability with the intelligent controller based on a Raspberry Pi and familiarity with the software language. An alternative software language option for the proposed work would be C/C++.

3.4.2 GitHub Version Control Tool

Although software backups can be kept locally on Raspberry Pi using the micro SD card, GitHub version control tool is also accommodated [54]. Therefore, produced software revisions can be kept online, providing a thorough software development flow and adding another layer of protection in terms of backup purposes.

3.4.3 VNC Viewer Tool

For the communication between the intelligent controller and the PC, VNC Viewer tool is used, allowing virtual communication, and thus, avoiding the utilization of an external screen in the system. VNC Viewer, a third-party tool, is available but not limited to Linux, macOS and Windows platforms and is coming pre-installed with Raspberry Pi [55]. VNC Viewer can provide secure remote access with the host PC, direct or via the cloud [56].

For the proposed work, direct connection between the intelligent controller based on a Raspberry Pi and the host PC is selected, running on the same private local network (LAN), through an Ethernet-to-Ethernet connection. Although a direct Ethernet-to-Ethernet connection provides a secure and fast remote communication, in some cases is not practical. In fact, Ethernet is primarily used for network connections, and thus, PCs usually have only one such available socket. Instead, a wider-acceptable solution of Universal Serial Bus (USB) can be used, where an Ethernet-to-USB adaptor must be accommodated. For this project, host PC has more than one available Ethernet sockets, and therefore, above variation was not compulsory.

Lastly, VNC Viewer was chosen over its competitors because is user-friendly, has minimal latency and is recommended by Raspberry Pi.

3.4.4 Lake Shore Communication Tool

Lake Shore communication utility tool is used for early experimentation and hardware unit verification/testing purposes. Through Lake Shore utility tool, successful communication with the gaussmeter (including Hall sensor) can be verified by serially sending/receiving different messages in the form of commands or queries [57]. Figure 3.10 illustrates Graphical User Interface (GUI) of Lake Shore communication utility tool.

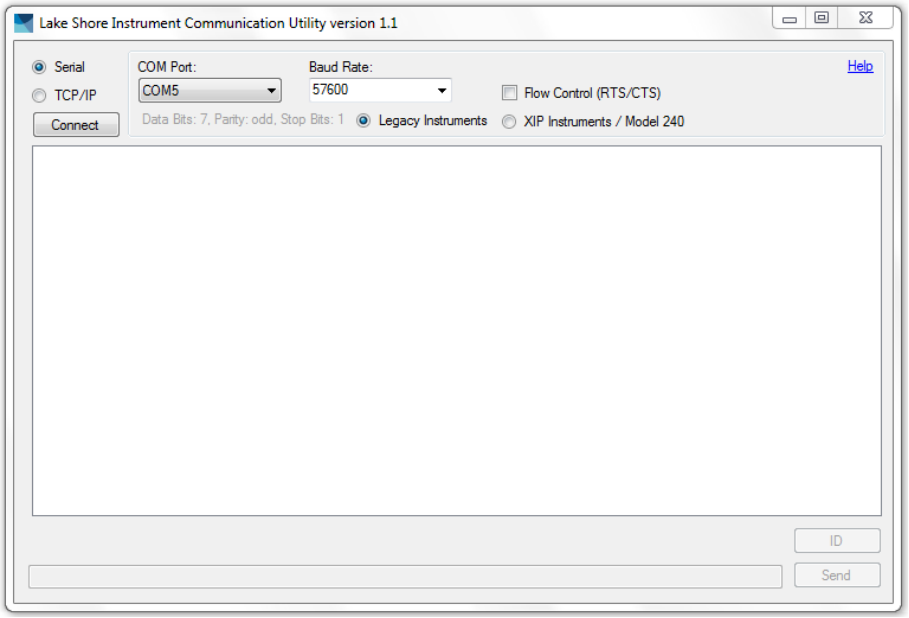


Figure 3.10: Graphical User Interface (GUI) of Lake Shore communication utility tool.

3.4.5 National Instruments VISA Interactive Control Tool

National Instruments Virtual Instrument Software Architecture (VISA) interactive control tool is used for early experimentation and hardware unit verification/testing purposes. At first instance, through VISA interactive control tool, initial GPIB interface with the power supply can be verified by connecting the USB wire-end of the NI GPIB-USB-HS control device to the host PC. GPIB wire-end of NI GPIB-USB-HS control device must then be connected to the Kepco's BOP power supply appropriate socket.

After establishing an initial GPIB interface, VISA interactive control tool can validate power supply remote communication by sending/receiving different SCPI (Standard Commands for Programmable Instruments) messages in the form of commands or queries [58, 59]. Finally yet importantly, if the above procedure is followed and successfully executed, it can consecutively verify that the built-in BIT 4886 digital interface card of Kepco BOP power supply is functioning properly. Figure 3.11 represents Graphical User Interface (GUI) of National Instruments VISA interactive control tool.

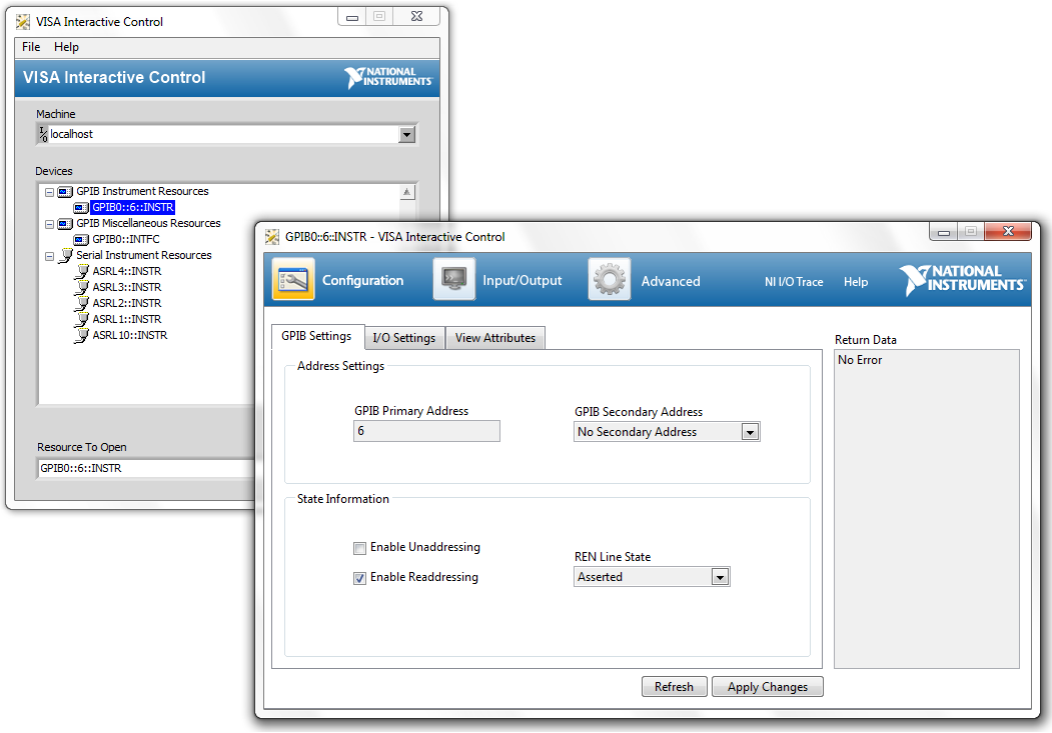


Figure 3.11: Graphical User Interface (GUI) of National Instruments VISA interactive control tool.

Chapter 4

DESIGN AND IMPLEMENTATION

After the specification of the hardware and software elements required for the proposed magnetic field control system, design (see section 4.1) and software implementation (see section 4.2) phases are followed in this chapter.

4.1 Design

A key driver impacting the overall quality of a control system is the design phase, prior to moving to the software implementation. The following sections discuss the system requirements specification (see section 4.1.1), modelling of the control system (see section 4.1.2) and high-level software architecture (see section 4.1.3), together with the appropriate testing and evaluation plans (see section 4.1.4).

4.1.1 System Requirements Specification

Specification and prioritization of functional, non-functional and usability requirements for the intelligent controller were an important aspect of the design phase, helping the overall management and execution of the project.

Functional requirements refer to the functions of the system including any inputs, outputs, data, calibration and processing features [60]. On the other hand, non-functional requirements can be described as the conditions that a system or a function of the system have to meet [60]. Generally, non-functional requirements give an answer to the question ”How does the system do it?” and have no semantic influence to the system or to a function of the system without the appearance of functional requirements [60]. Usability requirements are depending on how easy the system or a

functionality of the system can be anticipated by the user(s) [61].

MoSCoW Prioritization

For this project, classification and prioritization of system requirements were prepared using the "MoSCoW" technique, a common software engineering approach [62]. Each upper-case letter of "MoSCoW" abbreviation refers to a series of prioritization divisions as follows [62]:

- "**M**ust have" requirements for the successful demonstration of the final system prototype.
- "**S**hould have" requirements considered as high-priority that should be introduced to the final system prototype.
- "**C**ould have" requirements considered as desirable, but are not necessary for the basic functionality of the system.
- "**W**ould have" requirements considered as upgrades for a future version of the system.

Full system requirements classified and prioritized using the "MoSCoW" technique can be found in Appendix D-Table D.1.

4.1.2 Modelling the Control System

After the identification of system requirements, the proposed closed-loop (or feedback) control system had to be modelled. In the context of dynamic control, an intelligent controller based on a Raspberry Pi should be able to control and modify the magnetic field using a combination of both correction element and feedback signal. For this system, the power supply is referred to as the correction element, responsible for supplying the electrical current. It is important that magnetic field is a result of electrical current flowing in the coils of the electromagnet, and this is referred to as the process for this system. The feedback signal comprises of a constant comparison between the desired and actual magnetic fields, which are measured using the Hall sensor of the gaussmeter. Depending on the difference between the desired and the actual magnetic fields, the error signal in the system can be either positive or negative. Figure 4.1 shows a block diagram comprising of the modelled elements of the proposed magnetic field closed-loop (or feedback) control system.

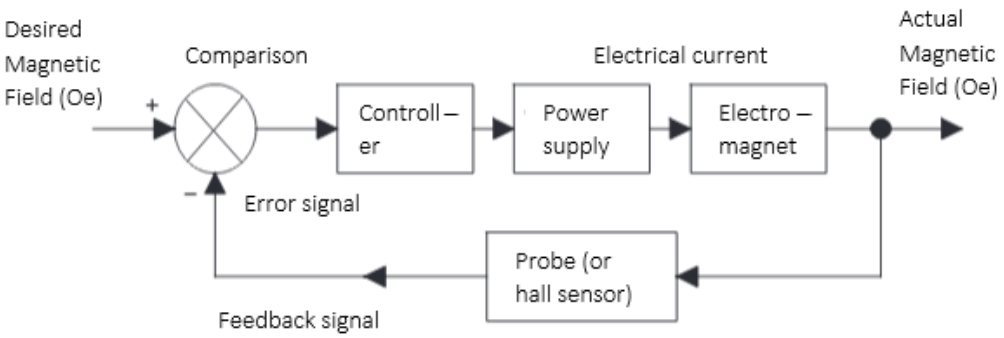


Figure 4.1: A block diagram comprising of the modelled elements of the proposed closed-loop (or feedback) magnetic field control system (diagram structure from [1]).

4.1.3 High-level Software Architecture

A use-case scenario of the proposed control system would start by prompting the user to input a desired magnetic field. Consecutively, the intelligent controller establishes a remote communication with the gaussmeter to read the actual magnetic field value using the Hall sensor. If the difference between the desired magnetic field input and the actual magnetic field reading is not zero, the intelligent controller applies a PID (Proportional-Integral-Derivative) method to determine the electrical current change needed to reach the desired value.

It is important that the magnetic field output is a result of the electrical current flowing in the coils of the electromagnet. Therefore, the intelligent controller establishes a remote communication with the power supply to request electrical current amendment that will eventually change the magnetic field. When the actual magnetic field reading reaches the required input prompted by the user, intelligent controller outputs the desire magnetic field and keeps checking until user inputs another desire magnetic field or terminates the program. Figure 4.2 shows the software flow diagram for the proposed magnetic field closed-loop (or feedback) control system.

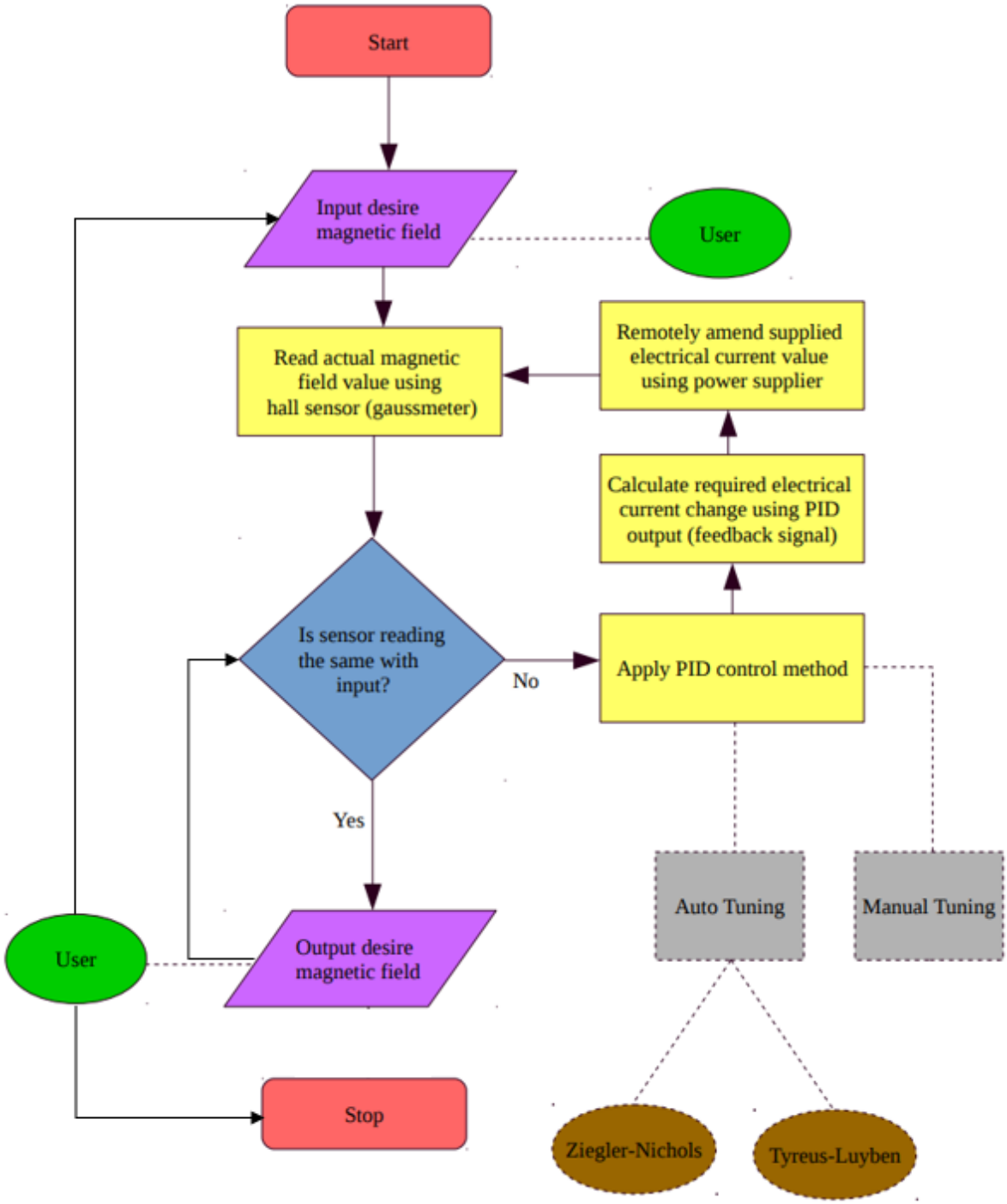


Figure 4.2: Software flow diagram for the proposed magnetic field closed-loop (or feedback) control system. Colorful shapes differentiate the activities that have to be executed by either the user, system or a component of the system.

UML Class Diagram

Software framework for intelligent controller system was set to be produced using Python, an object-oriented programming language. Considering that, initiation of class diagram during the design phase was a core requirement. Hence, structural and behavioural features, as well as, relationships between features (or classes) of the proposed software framework could be defined.

For this work, a class diagram was produced using Unified Modelling Language (UML) guidelines [63]. UML is a standard formalism for software design and analysis, allowing the static structure representation of an application or a system domain [63]. Figure 4.3 illustrates the UML class diagram for the proposed intelligent controller system. For simplicity purposes, at this stage, class diagram consists of class names only, avoiding any class attributes and functions (or operations).

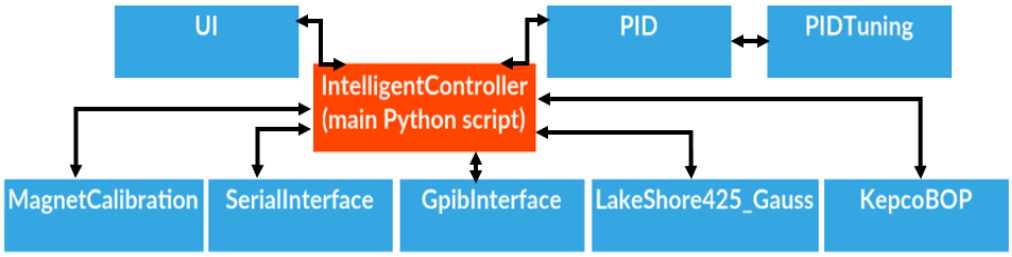


Figure 4.3: UML class diagram for the proposed intelligent controller system.

In particular, intelligent controller system comprises of *SerialInterface*, *GpibInterface*, *LakeShore425_Gauss*, *KepcoBOP*, *PID*, *PIDTuning*, *MagnetCalibration* and *UI* Python classes. Each class denotes an object which can be instantiated in the intelligent controller main Python script. Object of a class can also be instantiated in another class, where relationship between the two classes is pre-declared. Relationships between the classes are shown using bidirectional arrows. Finally, functions (or operations) of a class can be called from the intelligent controller main Python script or by another related class using the instantiated object.

4.1.4 Testing and Evaluation Plans

For drawing meaningful conclusions about the intelligent controller based on a Raspberry Pi system, a thorough testing and evaluation plans had to be set in place at early stages of the project.

Generally, in a software engineering project, testing is aiming to find any failed requirements, usability issues, functionality bugs and performance indices. For this project, ultimate goal of testing phase is achieving the required system functionality and usability along with the investigation and optimization (if possible) of stability, accuracy, speed and overshooting performance properties.

Hardware and software unit tests are set to be conducted for verifying and investigating individual components/elements behaviour in the system. Integrated system tests have also been identified in order to observe how individual elements of the system would fit together, and thus, test different use-case scenarios.

In the scope of integrated tests, for measuring and optimizing SASO (accuracy, stability, speed and overshooting) performance properties of the intelligent PID controller, different manual and auto tuning techniques are set to be tested and evaluated. Auto tuning includes some of the most commonly-used methods for closed-loop (or feedback) control system such as Ziegler-Nichols and Tyreus-Luyben [28, 29].

Table 4.1 shows unit and integrated testing/evaluation plans for the intelligent controller based on a Raspberry Pi system.

No.	Test type	Description
1	Unit (Hardware)	Test/verification of Lake Shore 425 gaussmeter initial communication by prompting different message strings (commands/queries) through Lake Shore communication tool.
2	Unit (Hardware)	Test/verification of Kepco BOP power supply initial communication by prompting different SCPI message strings (commands/queries) through NI VISA interactive control tool.
3	Unit (Software)	Validate desired magnetic field input format through terminal-based UI.
4	Unit (Software)	Validate any functionalities of the system, individually, through terminal-based UI.
5	Integrated	Test overall functionality of the control system by entering different magnetic field inputs through terminal-based UI (and GUI if available). Measure, optimize and evaluate SASO properties using different manual and auto tuning methods.

Table 4.1: Unit and integrated testing/evaluation plans for the intelligent controller based on a Raspberry Pi system.

Finally, yet importantly, intelligent controller system is set to be evaluated using the quantified assessment benchmarks of integrated tests. Graphs and summary tables will be initiated for results, tuning methods comparisons, performance properties and optimization discussion. Assessment of the system requirements and possible alternative technical approaches for the system design and implementation would also be considered, if appropriate.

4.2 Implementation

A custom-initiated software framework has been developed using Python programming language in order to control the magnetic field from an electromagnet, through the intelligent controller. Produced software supports other features as well, enhancing the overall quality of the proposed magnetic field control system. Considering the importance of software re-usability, object-oriented Python classes designed in a way that they can be utilized by an upgraded version of this project or any other project aiming to communicate with the specified instruments or interfaces. Full Python source code implemented for the intelligent controller based on a Raspberry Pi project can be viewed and retrieved from the author's GitHub page [64].

For the successful execution and monitoring of the project, software implementation has been divided into different technical milestones. The following discussion covers configurations and external Python software libraries (see section 4.2.1), along with input method (see section 4.2.2), controlling the gaussmeter (see section 4.2.3), controlling the power supply (see section 4.2.4), mathematical modelling of the process (see section 4.2.5), PID control technique (see section 4.2.6), magnet calibration feature (see section 4.2.7), UI (see section 4.2.8), final software integration (see section 4.2.9) and GUI (see section 4.2.10).

4.2.1 Configurations and External Python Software Libraries

Prior to moving to the actual software implementation, communication configurations and external software libraries had to be set and imported, accordingly. Configurations, and thus, establishment of individual communications between each instrument of the system with the intelligent controller were two of the most time-consuming technical tasks of the project. The following sub-sections discuss the communication establishment between the PC, gaussmeter, power supply and the Raspberry Pi together with

the external Python software libraries.

PC and Raspberry Pi Communication

As a compact computer board, Raspberry Pi operates through its own Linux-based (Raspbian or NOOBS) operating system (OS). Therefore, for an effective usage, Raspberry Pi should be connected to an external monitor. However, for this work, Raspberry Pi had to be connected to a host PC, considering that the principle of the project includes the future integration of this system to a broader one which utilizes a PC. To allow the control of the Raspberry Pi without an external monitor and directly from the host PC, a third-party software (e.g. SSH or VNC Viewer) was required.

For this project, VNC Viewer [55, 56], a third-party tool pre-installed in Raspberry Pi was used to establish a secure remote communication with the host PC, via an Ethernet-to-Ethernet private LAN (Local Area Network) connection. After installing the VNC Viewer tool on the host PC, it had to be configured on Raspberry Pi. From the Raspberry Pi's terminal, VNC Viewer can be enabled using the following command:

sudo raspi-config

This command pops-up Raspberry Pi's configuration page which includes the enablement of VNC feature. After obtaining the Raspberry Pi's static IP (Internet Protocol) address, VNC Viewer was ready to run on the host PC, establishing a virtual communication with the Raspberry Pi.

Gaussmeter and Raspberry Pi Communication

Next, configurations required for the successful communication between the gaussmeter and the Raspberry Pi. In order to establish a USB communication with the Lake Shore 425 gaussmeter, a set of interface parameters had to be configured. Table 4.2 shows the configuration of parameters required by the Lake Shore 425 gaussmeter for the successful serial interface.

Parameter	Configuration
Baud rate	57,600
Data bits	7
Start bits	1
Stop bits	1
Parity	Odd
Flow control	None
Handshaking	None

Table 4.2: Configuration of parameters required by the Lake Shore 425 gaussmeter for the successful serial interface [37].

In particular, above parameters configured in *__init__* function of *SerialInterface* Python class which will be discussed in section 4.2.3. As a pre-requisite, and for the effective serial communication between the Lake Shore 425 gaussmeter and the intelligent controller, external PySerial Python software library had to be imported in the project directory as well.

Power Supply and Raspberry Pi Communication

Kepco BOP power supply in conjunction with the National Instruments GPIB-USB-HS control device had to be configured at the software level in order to establish a thorough GPIB communication with the BIT 4886 digital interface card.

Although National Instruments provides the GPIB drivers for both Windows and Linux environments, it does not support Raspberry Pi's distributions (Raspbian or NOOBS) for Linux [65]. This is a common issue as we are approaching the era of replacing GPIB interface by other modern interfaces such as USB and Ethernet (or LAN) [66].

In order to manually install and configure the GPIB driver on Raspberry Pi's Linux-based operating system (OS) through the command line, the step-by-step tutorial of Tsemenko [66] was followed:

1. Updated current packets of Raspberry Pi as a root, using *sudo apt-get update* command.

2. Found the current kernel version of Raspberry Pi using *uname -a* command.

3. Built the headers for current kernel version (in this case 4.19.46-v7+) of Raspberry Pi using *rpi-source* command.

4. Installed dependencies, libraries and additional packages required by the GPIB driver using *apt-get install* command.

5. Retrieved the latest version of Linux GPIB driver from Source Forge Linux GPIB Support page [67].

6. Compiled and installed the Linux GPIB driver using *./bootstrap*, *./configure*, *make* and *make install* commands, in series.

7. Linked all installed libraries using *ldconfig* command.

8. Modified */etc/gpib.conf* file to point to the correct board type (for NI GPIB-USB-HS control device usage).

9. Loaded the kernel module as a root, using *sudo modprobe ni_usb_gpib* command.

10. Run *sudo gpib_config* command as a root to verify successful configuration of Linux GPIB driver on Raspberry Pi.

11. Added the ”gpib_config” path to the /etc/rc.local directory of Raspberry Pi in order to set-up GPIB communication automatically upon Raspberry Pi’s boot, without the need of executing *sudo gpib_config* command every time.

Final step was important considering that the proposed system has to avoid any side configurations in order to conform with the possibility of future integration to a wider system. Above procedure is only a general guidance on how to install and configure Linux GPIB driver on Raspberry Pi 3 Model B+. For any other Raspberry Pi model, procedure might have to be revised. Please refer to Tsemenko’s [66] tutorial for detailed explanation on how to set-up GPIB driver on Raspberry Pi’s Linux-based operating system (OS).

Conclusively, after configuring the GPIB driver on Raspberry Pi, external Gpib Python software library had to imported in the project directory as well. Therefore, using Gpib library’s built-in functions, GPIB communication could be established between the Kepco BOP power supply and the intelligent controller. More details regarding GPIB interface in the system can be found in section 4.2.4.

External Python Software Libraries

For the adequate implementation of the software, some external Python libraries were also required. A core advantage of Python programming language is that it offers numerous well-defined software libraries, serving different purposes. It is important that in order to retrieve and import external Python software libraries, Raspberry Pi had to temporarily access the internet.

In particular, time, datetime, math, matplotlib and Tkinter Python packages were imported to the project directory for managing the delays between different processes, showing real-time updates to the user, doing required maths conversions, plotting the control system trials and building the GUI (Graphical User Interface) for the system, respectively.

4.2.2 Input Method

Desired magnetic field input is gathered by using the *raw_input* function in *UI* Python class. Figure 4.4 illustrates the source code line of *UserInterface* function, prompting the user to enter the desired magnetic field.

```
121        self.desire_magnetic_field = raw_input ("Enter the Desired Magnetic " +
122                                                "Field (Oe): ")
```

Figure 4.4: Python *raw_input* function, prompting the user to enter the desired magnetic field.

It is important that *raw_input* function saves the assigned value in string format, and therefore, further conversion to float data type had to be considered in order to conform with system requirements.

More detailed discussion of user experience and further functionalities offered by the terminal-based UI is followed in section 4.2.8.

4.2.3 Controlling the Gaussmeter

After identifying the pre-requisites for communicating with the gaussmeter (see section 4.2.1), it was time to develop the software to do so. However, before implementing *SerialInterface* and *LakeShore425_Gauss* Python classes for controlling the gaussmeter (including Hall sensor), an in-depth understanding of Lake Shore message strings

formatting and general commands/queries transmission and reception rules was required.

The following sub-sections discuss Lake Shore's gaussmeter message strings formatting and commands/queries transmission and reception rules that has to be followed. Next, the development of both *SerialInterface* and *LakeShore425_Gauss* Python classes will be discussed.

Lake Shore Message Strings Formatting

In order to successfully establish a USB interface communication with the Lake Shore 425 gaussmeter, message strings must be formulated carefully [37].

Lake Shore 425 gaussmeter uses ASCII (American Standard Code for Information Interchange) character format [37]. Therefore, a group of ASCII characters can form a message string. For Lake Shore 425 instrument, there are three different types of message strings. Namely, commands, queries and responses [37].

Commands, as well as, queries can be issued through a user program (e.g. intelligent controller), while responses are the "answers" to queries handled by the Lake Shore 425 gaussmeter [37]. A command notifies the Lake Shore 425 gaussmeter to execute a function, while a query requires a response to be sent from the instrument to the user program [37].

The format of a command message string for Lake Shore 425 gaussmeter is as follows [37]:

$$< \textbf{command} >< \textbf{space} >< \textbf{parameters} >< \textbf{terminators} >$$

The format of a query message string for Lake Shore 425 gaussmeter is as follows [37]:

$$< \textbf{query} >< \textbf{?} >< \textbf{space}(\text{if any par.}) >< \textbf{parameters}(\text{if any}) >< \textbf{terminators} >$$

In both command and query cases, terminators have to be formatted and used correctly for the successful communication with the Lake Shore 425 instrument. In particular, terminators consist of an ASCII carriage return (CR) character followed by an ASCII linefeed (LF) character [37]. In a message string format, this is formulated as "\ r\ n", where "\ r" is the carriage return and "\ n" is the new line. Data formats of Lake Shore 425 gaussmeter response message strings vary and are solely depending on the query that it is being sent through the program. For instance, "RDGFIELD?\ r

\ n" query responses with the field reading in a format based on the specified range and unit. On the other hand, for example "TYPE?\ r \ n" query responses with the probe type in a double-digit format.

Conclusively, two or more commands and/or queries can be combined in a single communication between the program and the instrument by separating them with a semi-colon(;) [37].

Lake Shore Commands/Queries Transmission and Reception Rules

For a thorough communication between the user program (e.g. intelligent controller) and the Lake Shore 425 gaussmeter, a set of commands and queries transmission and reception rules have to be considered, respectively.

Commands transmission rules for a user program aiming to communicate with the Lake Shore 425 gaussmeter are listed as follows [37]:

- Command must be formatted and transmitted as one message string, including the terminators ("\ r \ n").
- Another communication must not be initiated earlier than 30 milliseconds after the last command is being transmitted.
- Communication frequency must not exceed 30 times per second.

Queries transmission rules for a user program aiming to communicate with the Lake Shore 425 gaussmeter are listed as follows [37]:

- Query must be formatted and transmitted as one message string, including the terminators ("\ r \ n").
- User program should expect an immediate response after a query is sent.
- User program should be prepared to receive the entire response, including the terminators ("\ r \ n").
- Another communication must not be initiated while waiting for the response or earlier than 30 milliseconds after the last query's response has been received.
- Communication frequency must not exceed 30 times per second.

Development of SerialInterface Python Class

Lake Shore 425 gaussmeter cannot initiate communication or guarantee timings between the message strings (commands or queries) [37]. This is a responsibility of the intelligent controller. Therefore, following the Lake Shore 425 gaussmeter transmission and reception rules, *SerialInterface* Python class has been implemented for USB (Universal Serial Bus) communication with the instrument.

After the configuration of serial interface through *__init__* function, another two functions have been produced for writing and reading commands and queries responses to/from Lake Shore 425 gaussmeter, respectively.

In particular, *SerialInterface* Python class consists of *serialFullCommunication* and *serialWriteOnly* functions. *serialFullCommunication* function sends a command or query to the Lake Shore 425 gaussmeter and reads instrument's response using PySerial library's functions. Figure 4.5 shows the *serialFullCommunication* function's source code of *SerialInterface* Python class.

```
26      ''' serialFullCommunication function writes and reads
27          a command/query using Python serial library.
28      # @param: command_query - enter command/query to write '''
29      def serialFullCommunication(self, command_query):
30          if self.serialInterface.isOpen():
31              try:
32                  self.serialInterface.flushInput()
33                  self.serialInterface.flushOutput()
34                  self.serialInterface.write(command_query +
35                                             self.messageTerminators)
36                  time.sleep(0.03)
37                  reading = ''
38                  while self.serialInterface.inWaiting():
39                      reading += self.serialInterface.read()
40                  return str(reading)
41              except Exception:
42                  raise Exception ("Serial communication error" +
43                                   " (cannot write/read).")
44          else:
45              raise Exception ("Cannot open serial port.")
46          self.serialInterface.close()
```

Figure 4.5: *serialFullCommunication* function's source code of *SerialInterface* Python class.

Furthermore, *serialWriteOnly* function has also been developed for *SerialInterface* Python class' purposes. *serialWriteOnly* function sends a command or query to the Lake Shore 425 gaussmeter using PySerial library's functions, without waiting for reading the instrument's response. Hence, *serialWriteOnly* function was useful at the final stage of the implementation, when response from Lake Shore 425 gaussmeter was not compulsory for commands, shorting the delay of intelligent controller's and instrument's communication. Figure 4.6 illustrates the *serialWriteOnly* function's source code of *SerialInterface* Python class.

```
48      ''' serialWriteOnly function writes
49          a command/query using Python serial library.
50      # @param: command_query - enter command/query to write '''
51      def serialWriteOnly(self, command_query):
52          if self.serialInterface.isOpen():
53              try:
54                  self.serialInterface.flushInput()
55                  self.serialInterface.flushOutput()
56                  writing = self.serialInterface.write(command_query +
57                                                       self.messageTerminators)
58                  time.sleep(0.03)
59                  return str(writing)
60              except Exception:
61                  raise Exception ("Serial communication error" +
62                                   " (cannot write/read).")
63          else:
64              raise Exception ("Cannot open serial port.")
65          self.serialInterface.close()
```

Figure 4.6: *serialWriteOnly* function's source code of *SerialInterface* Python class.

For a thorough communication between the intelligent controller and the Lake Shore 425 gaussmeter, it is vital that both *serialFullCommunication* and *serialWriteOnly* functions include the appropriate message terminators ("\ r \ n") when writing a command/query. Exceptions for handling any possible errors during the communication have also been accommodated in both *serialFullCommunication* and *serialWriteOnly* functions. Table 4.3 briefly describes the functions developed for *SerialInterface* Python class.

Function	Description	Parameter(s)	Returns
__init__	Establishes initial serial communication with ttyUSB0 using PySerial library's appropriate function. Constructs message terminators that are being used from other functions of *SerialInterface* class.	-	-
serialFullCommunication	Writes and reads a command/query using PySerial library's appropriate functions.	command_query	str(reading)
serialWriteOnly	Writes a command/-query using PySerial library's appropriate functions.	command_query	str(writing)

Table 4.3: Functions developed for *SerialInterface* Python class.

Finally, yet importantly, *SerialInterface* Python class implemented in a way that it can be utilized with any instrument aiming to establish a USB communication.

Development of LakeShore425_Gauss Python Class

For a thorough communication between the Lake Shore 425 gaussmeter and the intelligent controller, *LakeShore425_Gauss* Python class had to be introduced. *LakeShore425_Gauss* Python class is responsible for formatting the message string (command or query) which is then being called by using *SerialInterface's* class function(s).

For this work, "UNIT", "RANGE" and "RDGFIELD?" message strings of Lake Shore 425 gaussmeter were compulsory. In particular, "UNIT 3" command used to configure the magnetic field unit, where 3 indicates the Oersted field unit. Figure 4.7 shows the *c_FieldUnits* function's source code which is used to format and return "UNIT 3" command as a message string.

```
''' c_FieldUnits command function.
# @param: units - enter units in int format '''
def c_FieldUnits(self, units):
    if (units == str(units) or units != int(units)):
        raise TypeError ("Enter a positive whole number (either" +
                         " 1-Gauss, 2-Tesla, 3-Oersted or 4-Ampere/meter)" +
                         " for field units parameter." +
                         " Value entered: {}. ".format(units) +
                         "Refer to help function for guidance of usage.")
    elif (units > self.UNITS_MAX or units < self.UNITS_MIN):
        raise ValueError ("Enter a positive whole number (either" +
                          " 1-Gauss, 2-Tesla, 3-Oersted or 4-Ampere/meter)"+
                          " for field units parameter." +
                          " Value entered: {}. ".format(units) +
                          "Refer to help function for guidance of usage.")
    else:
        return self.unit_command + str(units)
```

Figure 4.7: *c_FieldUnits* function's source code of *LakeShore425_Gauss* Python class.

In addition, "RANGE 2" command used to configure the magnetic field range, where 2 specifies the selected range. It is worthwhile mentioning that, field range should be specified based on the probe's (or Hall sensor) type. For this project, High Sensitivity Probe (HSE) was used. Figure 4.8 illustrates the *c_FieldRange* function's source code which is used to formulate and return "RANGE 2" command as a message string.

```
''' c_FieldRange command function.
# @oaram: field_range - enter field range in int format'''
def c_FieldRange(self, field_range):
    if (field_range == str(field_range) or field_range != int(field_range)):
        raise TypeError ("Enter a positive whole number (1-4) for" +
                         " field range parameter." +
                         " Value entered: {}. ".format(field_range) +
                         "Refer to help function for guidance of usage.")
    elif (field_range > self.FIELD_RANGE_MAX or
          field_range < self.FIELD_RANGE_MIN):
        raise ValueError ("Enter a positive whole number (1-4) for" +
                          " field range parameter." +
                          " Value entered: {}. ".format(field_range) +
                          "Refer to help function for guidance of usage.")
    else:
        return self.range_command + str(field_range)
```

Figure 4.8: *c_FieldRange* function's source code of *LakeShore425_Gauss* Python class.

Moreover, ”RDGFIELD?” query has been accommodated to read the magnetic field value using the probe (or Hall sensor). Figure 4.9 demonstrates the *q_FieldReading* function’s source code which is used to format and return ”RDGFIELD?” query as a message string.

```
439     ''' q_FieldReading query function. '''
440     def q_FieldReading(self):
441         return self.rdgfield_query
```

Figure 4.9: *q_FieldReading* function’s source code of *LakeShore425_Gauss* Python class.

Since the response of Lake Shore 425 gaussmeter regarding ”RDGFIELD?” query is in scientific form (+-nnn.nnnE+-nn), parsing was required for further processing and readability purposes. Parsing of magnetic field reading, and thus, conversion to a float value was handled in intelligent controller main Python script (see section 4.2.9) by simply using *float* function of Python.

Table 4.4 briefly describes the functions of *LakeShore425_Gauss* Python class used for this project. Pre-fix c_ indicates a function returning a command message string, while q_ pre-fix indicates a function returning a query message string.

Function	**Description**	**Parameter(s)**	**Returns**
__init__	Constructs all the Lake Shore message strings that are going to be used from other functions of *LakeShore425_Gauss* Python class. Parameter variables are also being constructed.	-	-
c_FieldUnits	Formats and returns ”UNIT units” command as a message string.	units	self.unit_command + str(units)

c_FieldRange	Formats and returns "RANGE field_range" command as a message string.	field_range	self.range_command + str(field_range)
q_FieldReading	Formats and returns "RDGFIELD?" query as a message string.	-	self.rdgfield_query

Table 4.4: Functions of *LakeShore425_Gauss* Python class used for this system.

Furthermore, it is important that both *c_FieldUnits* and *c_FieldRange* functions include appropriate type and value exceptions for validating the correct message formatting of a string. On the other hand, query message string of *q_FieldReading* function ("RDGFIELD?") is pre-defined as it is not requiring the specification of any parameter. Therefore, validation using exceptions was redundant for *q_FieldReading* function.

For this system, utilization of just *c_FieldUnits*, *c_FieldRange* and *q_FieldReading* functions was convenient in order to gather required data from the Lake Shore 425 gaussmeter. However, for future usage purposes, all the available message strings (commands and queries) offered by the Lake Shore 425 gaussmeter have been included in the *LakeShore425_Gauss* Python class, forming a thorough library. Therefore, *LakeShore425_Gauss* Python class can be used for either an upgraded version of this project or any other project that is aiming to communicate with the Lake Shore 425 instrument.

Appendix E-Table E.1 states all the available functions of custom-initiated *LakeShore425_Gauss* Python class, each forming a dedicated message string (command or query).

4.2.4 Controlling the Power Supply

After the development of the software to control the Lake Shore 425 gaussmeter (including Hall sensor), it was time to implement the software to remotely control the Kepco BOP power supply. However, before producing *GpibInterface* and *KepcoBOP* Python classes for controlling the power supply, an in-depth understanding of Standard

Commands for Programmable Instruments (SCPI) message strings formatting was required. SCPI is a programming language conforming to the standards and protocols established by IEEE 488.2, offered by BIT4886 digital interface card which is included in Kepco BOP power supply [40].

The following sections discuss the Kepco's BOP SCPI message strings formatting and the development of both *GpibInterface* and *KepcoBOP* Python classes that will help control the power supply remotely.

SCPI Message Strings Formatting

In order to successfully establish a GPIB interface communication with the Kepco BOP power supply, message strings must be formatted correctly. Kepco BOP power supply uses SCPI (Standard Commands for Programmable Instruments) messages [40]. In particular, there are two different types of SCPI message strings. Namely, program and response messages [40]. Program messages are sent from the controller to the power supply, while response messages are sent from the power supply to the controller [40].

For a thorough communication between the intelligent controller and the Kepco BOP power supply, program, as well as, response SCPI messages should be handled by the software. SCPI program messages can be further divided to either commands or queries [40]. A SCPI program message in the form of command would notify the Kepco BOP power supply to execute a function, while a query would require a response to be sent from the Kepco BOP power supply to the intelligent controller [40].

For example, a SCPI program message in the form of query would ask the Kepco BOP power supply to read the electrical current. Kepco BOP power supply would answer back with the supplied electrical current in the form of SCPI response message.

The format of a SCPI program message string in the form of command for Kepco BOP supply is as follows [40]:

$$:< \textbf{command} >< \textbf{space} >< \textbf{parameters} >< \textbf{terminator} >$$

The format of a SCPI program message string in the form of query for Kepco BOP power supply is as follows [40]:

$$:< \textbf{query} >< ? >< \textbf{space}(\text{if any par.}) >< \textbf{parameters}(\text{if any}) >< \textbf{terminator} >$$

where colon (:) is the optional root specifier.

In both command and query cases, terminator has to be formatted and used correctly for the successful remote communication with the Kepco BOP instrument. Particularly, terminator consists of an ASCII linefeed (LF) character [40]. In a SCPI message string format, ASCII linefeed (LF) would be represented as "\ n".

Finally, yet importantly, two or more SCPI program messages in the form of either commands or queries can be combined in a single communication [40]. This can be done by separating them with a semi-colon (;), followed by the root specifier (:). It is important that root specifier (:) is required when two or more SCPI messages are formulated to be sent in a single communication, while it is optional when a single SCPI message (command or query) is formatted for transmission. An example of a SCPI program message string which is remotely setting the voltage level of Kepco BOP power supply to 5V and turning on the output using a single communication would be:

"VOLT 5;:OUTP 1\ n"

Development of GpibInterface Python Class

Kepco BOP power supply cannot initiate communication or guarantee timings between the SCPI message strings (commands or queries). Intelligent controller, and specifically Raspberry Pi is responsible for that. Hence, following the SCPI message strings formatting rules, *GpibInterface* Python class has been developed for GPIB (General Purpose Interface Bus) communication with Kepco BOP power supply.

After the configuration of GPIB interface through *__init__* function, two additional functions have been implemented for writing and reading SCPI message commands and queries responses to/from Kepco BOP power supply, accordingly.

Particularly, *GpibInterface* Python class comprises of *gpibFullCommunication* and *gpibWriteOnly* functions. *gpibFullCommunication* transmits a command or query to the Kepco BOP power supply and reads the instrument's response using Gpib library's functions. Figure 4.10 illustrates the *gpibFullCommunication* function's source code of *GpibInterface* Python class.

```
17      ''' gpibFullCommunication function writes and
18          reads a command/query using Python Gpib library.
19      # @param: command_query - enter command/query to write '''
20      def gpibFullCommunication(self, command_query):
21          try:
22              self.gpibInterface.write(command_query + self.messageTerminator)
23              time.sleep(0.03)
24              reading = self.gpibInterface.read(100)
25              return str(reading)
26          except Exception:
27              raise Exception ("Gpib communication error" +
28                               " (cannot write/read).")
```

Figure 4.10: *gpibFullCommunication* function's source code of *GpibInterface* Python class.

Moreover, *gpibWriteOnly* function has been produced for *GpibInterface* Python class' purposes. In fact, *gpibWriteOnly* function transmits a command or query to the Kepco BOP power supply using Gpib library's functions, without requiring instrument's response. *gpibWriteOnly* function was convenient at the final stage of implementation, when reading the printed response from Kepco power supply was not compulsory, especially for SCPI message commands. Therefore, delay of intelligent controller's and instrument's communication could be be shorten at a satisfactory level. Figure 4.11 shows the *gpibWriteOnly* function's source code of *GpibInterface* Python class.

```
30      ''' gpibWriteOnly function writes a command/query using
31          Python Gpib library.
32      # @param: command_query - enter command/query to write '''
33      def gpibWriteOnly(self, command_query):
34          try:
35              writing = self.gpibInterface.write(command_query +
36                                                 self.messageTerminator)
37              time.sleep(0.03)
38              return str(writing)
39          except Exception:
40              raise Exception ("Gpib communication error" +
41                               " (cannot write/read).")
```

Figure 4.11: *gpibWriteOnly* function's source code of *GpibInterface* Python class.

It is important that both *gpibFullCommunication* and *gpibWriteOnly* functions include the appropriate SCPI message terminator (”\ n”) for the successful communication with the Kepco BOP power supply. Exceptions for handling any potential errors during the communication between the intelligent controller and the Kepco BOP power supply have also been introduced in both *gpibFullCommunication* and *gpibWriteOnly* functions. Table 4.5 briefly describes the functions of *GpibInterface* Python class.

Function	**Description**	**Parameter(s)**	**Returns**
__init__	Establishes initial GPIB communication with address (0,6) using Python Gpib library's function. Constructs message terminator that is being used from other functions of *GpibInterface* class.	-	-
gpibFullCommunication	Writes and reads a command/query using Python Gpib library's functions.	command_query	str(reading)
gpibWriteOnly	Writes a command/-query using Python Gpib library's functions.	command_query	str(writing)

Table 4.5: Functions developed for *GpibInterface* Python class.

Conclusively, *GpibInterface* Python class developed in a way that it can be used with any other instrument seeking to establish a GPIB communication.

Development of KepcoBOP Python Class

For a comprehensive communication between the Kepco BOP power supply and the intelligent controller, *KepcoBOP* Python class had to be initiated. *KepcoBOP* Python class is responsible for formulating the SCPI message string (command or query) which is then being called by *GpibInterface's* class function(s).

For this closed-loop (or feedback) control system, "SYST:REM", "VOLT", "CURR", "FUNC:MODE" and "OUTP" SCPI message strings were mandatory. Especially, "SYST:REM 1" command used to establish the remote communication between the Kepco BOP power and the intelligent controller, where 1 sets the remote communication on. Figure 4.12 illustrates the *c_SetRemoteCommunication* function's source code which is used to format and return "SYST:REM 1" command as a SCPI message string.

```
1137      ''' c_SetRemoteCommunication command function. '''
1138      def c_SetRemoteCommunication(self, off_on):
1139          if (off_on == str(off_on) or off_on != int(off_on)):
1140              raise TypeError ("Enter a whole positive number (either 1-on or"+
1141                               " 0-off) for remote comm. off_on parameter."+
1142                               " Value entered: {}. "
1143                               .format(off_on) +
1144                               "Refer to help function for guidance of usage.")
1145          elif (off_on > self.ON or off_on < self.OFF):
1146              raise ValueError ("Enter a whole positive number (either 1-on or"+
1147                                " 0-off) for remote comm. off_on" +
1148                                " parameter." +
1149                                " Value entered: {}. "
1150                                .format(off_on) +
1151                                "Refer to help function for guidance of usage.")
1152          else:
1153              return self.syst_rem_command + str(off_on)
```

Figure 4.12: *c_SetRemoteCommunication* function's source code of *KepcoBOP* Python class.

Additionally, "VOLT v" command used to remotely set the voltage level of Kepco BOP power supply, where v is the set voltage value. Figure 4.13 shows the *c_SetVoltageLevel* function's source code which is used to formulate and return "VOLT v" command as a SCPI message string.

```
898     ''' c_SetVoltageLevel command function. '''
899     def c_SetVoltageLevel(self, voltage_value):
900         if (voltage_value != str(voltage_value)):
901             raise TypeError ("Enter digits with decimal point and Exponent," +
902                              " e.g. 2.71E1 for 27.1." +
903                              " Voltage value entered: {}. "
904                              .format(voltage_value) +
905                              "Refer to help function for guidance of usage.")
906         else:
907             return self.volt_command + voltage_value
```

Figure 4.13: *c_SetVoltageLevel* function's source code of *KepcoBOP* Python class.

Similarly, "CURR c" command used to remotely specify the electrical current level of Kepco BOP power supply, where c is the specified electrical current value. Figure 4.14 shows the *c_SetCurrentLevel* function's source code which is used to format and return "CURR c" command as a SCPI message string.

```
785     ''' c_SetCurrentLevel command function. '''
786     def c_SetCurrentLevel(self, current_value):
787         if (current_value != str(current_value)):
788             raise TypeError ("Enter digits with decimal point and Exponent," +
789                              " e.g. 2.71E1 for 27.1." +
790                              " Current value entered: {}. "
791                              .format(current_value) +
792                              "Refer to help function for guidance of usage.")
793         else:
794             return self.curr_command + str(current_value)
```

Figure 4.14: *c_SetCurrentLevel* function's source code of *KepcoBOP* Python class.

Furthermore, "FUNC:MODE mode" command used to remotely configure the operating mode of Kepco BOP power supply, where mode can either set to "VOLT" or "CURR" for voltage or current mode, respectively. Figure 4.15 shows the *c_OperatingMode* function's source code which is used to format and return "FUNC:MODE VOLT/CURR" command as a SCPI message string.

```
545     ''' c_OperatingMode command function. '''
546     def c_OperatingMode(self, mode):
547         if (mode == self.VOLT_STRING or mode == self.CURR_STRING):
548             return self.func_mode_command + mode
549         else:
550             raise ValueError ("Enter either VOLT (voltage) or CURR (current)" +
551                               " for operating mode parameter." +
552                               " Value entered: {}. "
553                               .format(mode) +
554                               "Refer to help function for guidance of usage.")
```

Figure 4.15: *c_OperatingMode* function's source code of *KepcoBOP* Python class.

Moreover, "OUTP 1" command used to remotely enable the output of Kepco BOP power supply, where 1 enables the output. Figure 4.16 illustrates the *c_EDOutput* function's source code which is used to formulate and return "OUTP 1" command as a SCPI message string.

```
522     ''' c_EDOutput command function. '''
523     def c_EDOutput(self, off_on):
524         if (off_on == str(off_on) or off_on != int(off_on)):
525             raise TypeError ("Enter a positive whole number " +
526                              "(either 1-on or 0-off) for power supplier" +
527                              " output off_on parameter."
528                              " Value entered: {}. "
529                              .format(off_on) +
530                              "Refer to help function for guidance of usage.")
531         elif (off_on > self.ON or off_on < self.OFF):
532             raise ValueError ("Enter a positive whole number " +
533                               "(either 1-on or 0-off) for power supplier" +
534                               " output off_on parameter."
535                               " Value entered: {}. "
536                               .format(off_on) +
537                               "Refer to help function for guidance of usage.")
538         else:
539             return self.outp_command + str(off_on)
```

Figure 4.16: *c_EDOutput* function's source code of *KepcoBOP* Python class.

As previously discussed, SCPI programming language gives the opportunity of sending two or more message strings (commands or queries) in a single communication. Therefore, *sendMultipleKepcoSCPI* function implemented which is used to formulate and return two or more SCPI message strings (commands or queries) as one,

following the SCPI message strings formatting rules. Figure 4.17 shows the *sendMultipleKepcoSCPI* function's source code. In particular, *sendMultipleKepcoSCPI* function was used to combine and return operating mode and output commands as one SCPI message string ("FUNC:MODE CURR;:OUTP 1").

```
1185      ''' sendMultipleKepcoSCPI queries/commands function. '''
1186      def sendMultipleKepcoSCPI(self, *args):
1187          if (len(args) <= self.MIN_SCPI_MSGS):
1188              raise Exception ("Enter at least 2 SCPI messages" +
1189                               " (commands/queries) in string ('') format." +
1190                               " Value entered: {}. ".format(args) +
1191                               "Refer to help function for guidance of usage.")
1192          else:
1193              split_string = self.message_unit_separator + self.root_specifier
1194              return split_string.join(args)
```

Figure 4.17: *sendMultipleKepcoSCPI* function's source code of *KepcoBOP* Python class.

For initial experimenting purposes only, SCPI message strings (commands or queries) "MEAS:VOLT?", "MEAS:CURR?", "*RST" and "*IDN" have also been used for measuring the voltage level, measuring the electrical current level, resetting and identifying the Kepco BOP power supply, accordingly.

Table 4.6 outlines the functions of *KepcoBOP* Python class used for this system. Pre-fix c_ indicates a function returning a command message string.

Function	**Description**	**Parameter(s)**	**Returns**
__init__	Constructs all the SCPI message strings that are going to be used from other functions of *KepcoBOP* Python class. Parameter variables are also being constructed.	-	-
c_SetRemote Communication	Formats and returns "SYST:REM off_on" command as a SCPI message string.	off_on	self.syst_rem_command + str(off_on)

c_SetVoltageLevel	Formats and returns "VOLT voltage_value" command as a SCPI message string.	voltage_value	self.volt_command + voltage_value
c_SetCurrentLevel	Formats and returns "CURR current_value" command as a SCPI message string.	current_value	self.curr_command + current_value
c_OperatingMode	Formats and returns "FUNC:MODE mode" command as a SCPI message string.	mode	self.func_mode_command + mode
c_EDOutput	Formats and returns "OUTP off_on" command as a SCPI message string.	off_on	self.outp_command + str(off_on)
sendMultiple KepcoSCPI	Formats and returns two or more SCPI message strings (commands or queries) as one, following the SCPI message strings formatting rules.	*args	split_string.join(args)

Table 4.6: Functions of *KepcoBOP* Python class used for this system.

It is important that *c_SetRemoteCommunication*, *c_SetVoltageLevel*, *c_SetCurrentLevel*, *c_OperatingMode*, *c_EDOutput* and *sendMultipleKepcoSCPI* functions include appropriate type and value exceptions for validating the correct message formatting of a string.

For this work, utilization of just *c_SetRemoteCommunication*, *c_SetVoltageLevel*, *c_SetCurrentLevel*, *c_OperatingMode*, *c_EDOutput* and *sendMultipleKepcoSCPI* command functions was convenient in order to remotely modify the electrical current supplied to the coils of the electromagnet, and thus, control the magnetic field output. However, for future usage purposes, all the available SCPI message strings (commands

and queries) offered by the built-in BIT 4886 digital interface card of Kepco BOP power supply have been implemented in *KepcoBOP* Python class, forming a thorough software library. Hence, *KepcoBOP* Python class can be used for either an upgraded version of this project or any other project that is planning to communicate with the KepcoBOP instrument.

Appendix F-Table F.1 states all the available functions of custom-initiated *KepcoBOP* Python class, each forming a dedicated SCPI message string (command or query).

4.2.5 Mathematical Modelling of the Process

Transfer function is a common way of mathematically modelling the process of a closed-loop (or feedback) control system (see section 2.3.3). However, in order to conduct transfer function principle, direct access to the input or output of the system is required.

For this work, the magnetic field input/output is a result of electrical current flowing in the coils of the electromagnet. Therefore, correlation between the controlled variable (electrical current) and the input/output (magnetic field) of the proposed control system could be obtained by matching controlled variable-to-input/output correspondents, using look-up tables technique.

The following sub-sections discuss the look-up tables technique conducted for the purposes of this project and the relationship between the electrical current and the magnetic field.

Look-up Tables Technique

Prior to conducting look-up tables technique, it was important to observe physical limitations of the proposed control system. In fact, Kepco BOP power supply is capable of supplying +-20 Amps electrical current. However, due to relatively small coils of the proposed electromagnet, supplied electrical current should not exceed +-3 Amps.

Considering that, look-up tables method manually conducted to find the linear correlation between the electrical current (Amps) and the magnetic field (Oersted), bounded on +-3 Amps. For more accurate correlation between the electrical current and the magnetic field, look-up tables technique conducted for both descending and ascending changes, with 0.25 Amps precision level. Tables 4.7 and 4.8 show the look-up tables method conducted for finding the electrical current (Amps) and magnetic field

(Oe) linear correlation for both descending and ascending changes.

Electrical Current(Amps)	**Magnetic Field(Oe)**
3	847
2.75	777
2.5	707
2.25	637
2	567
1.75	496
1.5	426
1.25	355
1	284
0.75	214
0.5	143
0.25	72
0	1
-0.25	-70
-0.5	-140
-0.75	-211
-1	-281
-1.25	-352
-1.5	-422
-1.75	-493
-2	-563
-2.25	-634
-2.5	-704
-2.75	-774
-3	-844

Table 4.7: Look-up table states the electrical current (Amps) and magnetic field (Oe) linear correlation for descending changes.

Electrical Current(Amps)	Magnetic Field(Oe)
-3	-844
-2.75	-774
-2.5	-704
-2.25	-634
-2	-564
-1.75	-493
-1.5	-423
-1.25	-352
-1	-282
-0.75	-211
-0.5	-141
-0.25	-70
0	1
0.25	72
0.5	143
0.75	213
1	284
1.25	355
1.5	425
1.75	496
2	566
2.25	636
2.5	707
2.75	777
3	847

Table 4.8: Look-up table states the electrical current (Amps) and magnetic field (Oe) linear correlation for ascending changes.

Transforming look-up tables raw data into a graphical representation, Figures 4.18 and 4.19 illustrate linear correlation between the electrical current (Amps) and the magnetic field (Oe) for both descending and ascending changes.

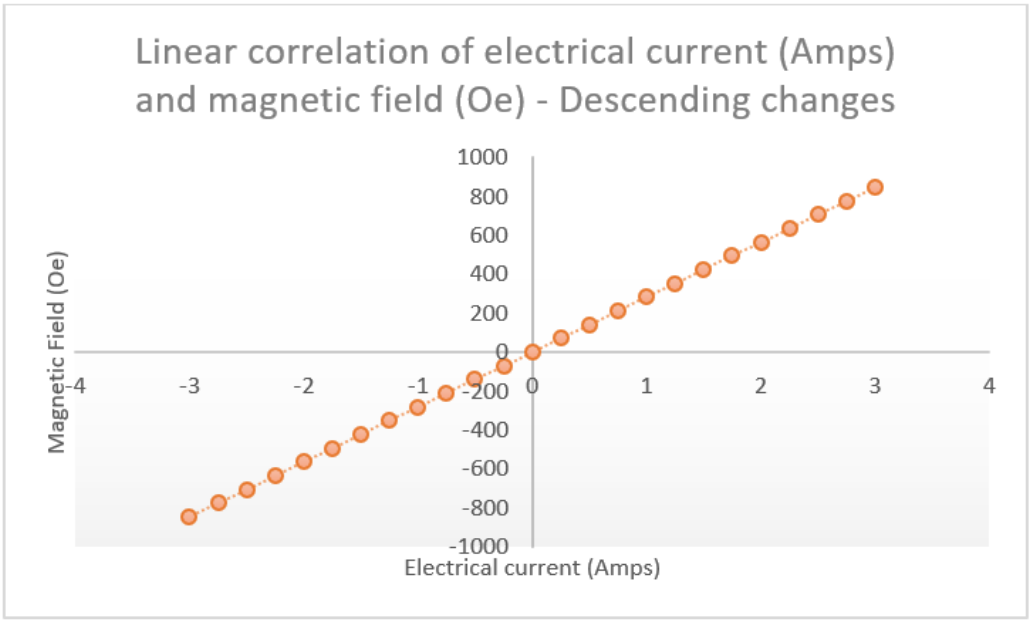


Figure 4.18: Graphical representation of electrical current (Amps) and magnetic field (Oe) linear correlation for descending changes.

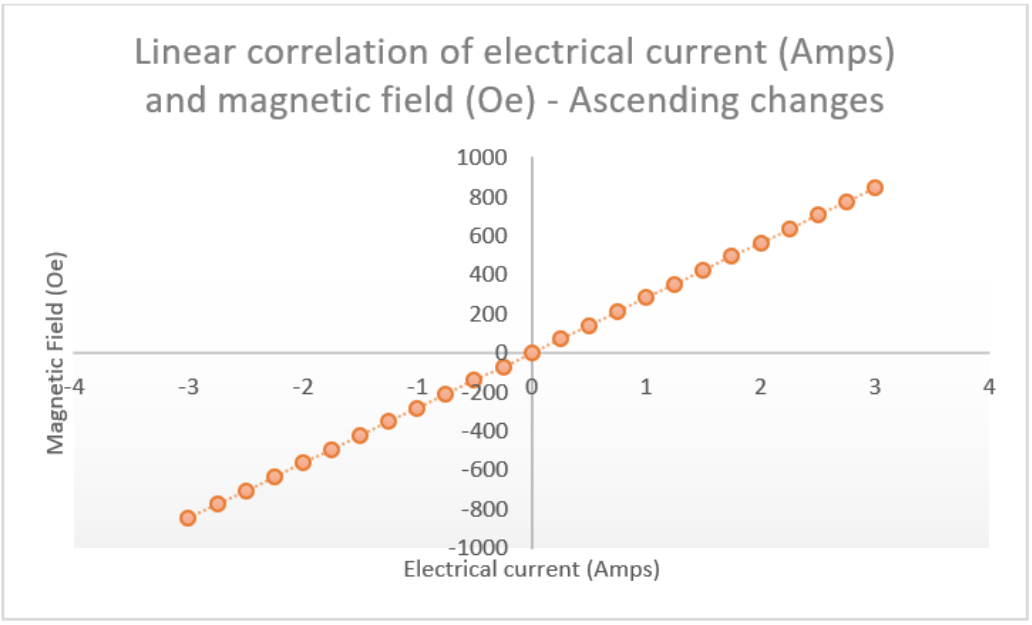


Figure 4.19: Graphical representation of electrical current (Amps) and magnetic field (Oe) linear correlation for ascending changes.

Analyzing Electrical Current and Magnetic Field Relationship

Analyzing both look-up tables (see Tables 4.7 and 4.8) and graphical (see Figures 4.18 and 4.19) data, it can be observed that there are small differences between descending and ascending linear correlations. Therefore, two different equations have been introduced for each descending (see equations 4.1 and 4.2) and ascending (see equations 4.3 and 4.4) changes in order to calculate the electrical current (Amps) and the magnetic field (Oe) values.

$$Magnetic\_Field = b\_descending * Electrical\_Current + a\_descending, \tag{4.1}$$

so

$$Electrical\_Current = \frac{Magnetic\_Field - a\_descending}{b\_descending}, \tag{4.2}$$

where a_descending calculated to $\sim$1.52 and b_descending calculated to $\sim$282.25, using Mathportal [68] correlation and regression calculator.

$$Magnetic\_Field = b\_ascending * Electrical\_Current + a\_ascending, \tag{4.3}$$

so

$$Electrical\_Current = \frac{Magnetic\_Field - a\_ascending}{b\_ascending}, \tag{4.4}$$

where a_ascending calculated to $\sim$1.20 and b_ascending calculated to $\sim$282.23, using Mathportal [68] correlation and regression calculator.

Finally, yet importantly, above equations along with a_descending, b_descending, a_ascending and b_ascending constants would only work for the proposed electromagnet which has electrical current limits of +-3 Amps. For instance, if a different electromagnet proposed with electrical current limits of +-5 Amps, a_descending, b_descending, a_ascending and b_ascending pre-calculated constants would not be applicable. Therefore, an automated magnet calibration feature introduced to the system

in order to handle different magnets. Detailed discussion of magnet calibration feature is followed in section 4.2.7.

4.2.6 PID Control Technique

Simply lowering or raising the supplied electrical current by a constant value to scale down the error signal between the desired and the actual magnetic fields would not comply with the fundamental principle of the project. In the context of a closed-loop control system, desired output (magnetic field) should be dynamically maintained using feedback signal, until the error signal has been eliminated. One common method of introducing such closed-loop (or feedback) control system is by accommodating PID (Proportional-Integral-Derivative) process control. The following discussion includes the development of both *PID* and *PIDTuning* Python classes.

Development of PID Python Class

Custom-initiated *PID* Python class is responsible for computing the controlled output, in the form of feedback signal, based on the elapsed time, error signal and pre-set proportional (Kp), integral (Ki) and derivative (Kd) gains. It is important that not all automated closed-loop control systems are making use of all three gain terms (Kp, Ki and Kd). This is solely depending on PID tuning which will be discussed in section 4.2.6.

After the preparation of PID controller through *__init__* and *SetUp* functions, another nine functions implemented for the adequate PID operation. In particular, *PID* Python class comprises of *SetKp*, *SetKi*, *SetKd*, *GetKp*, *GetKi*, *GetKd*, *SetPrevErr*, *SetWindup*, and *GenerateOutput* functions.

GenerateOutput function performs a PID computation and returns a control value, in the form of feedback signal, based on the elapsed time, error signal and the pre-set proportional (Kp), integral (Ki) and derivative (Kd) gains. In order to avoid ”integrator windup” issue, and thus, eliminate any possible overshooting in the proposed system, ”anti-windup” feature in the form of pre-determined set-point guards introduced in *GenerateOutput* function. Figure 4.20 illustrates the *GenerateOutput* function’s source code of *PID* Python class.

```
79      ''' Performs a PID computation and returns a control value based
80          on the elapsed time and the error signal.
81      # @param: error signal
82      # @return: PID control value '''
83      def GenerateOutput(self, error_signal):
84          self.current_time = time.time()
85          dt = self.current_time - self.previous_time
86          de = error_signal - self.previous_error_signal
87          self.Cp = self.Kp * error_signal
88          self.Ci += error_signal * dt
89
90          ''' Add windup function in order to eliminate overshooting. '''
91          if (self.Ci < -self.windup_guard):
92              self.Ci = -self.windup_guard
93          elif (self.Ci > self.windup_guard):
94              self.Ci = self.windup_guard
95
96          self.Cd = 0
97
98          ''' Ensuring no division by 0. '''
99          if dt > 0:
100             self.Cd = de/dt
101
102         self.previous_time = self.current_time
103         self.previous_error_signal = error_signal
104
105         ''' Sum all the PID terms and return the result. '''
106         return self.Cp + (self.Ki * self.Ci) + (self.Kd * self.Cd)
```

Figure 4.20: *GenerateOutput* function's source code of *PID* Python class.

In fact, *GenerateOutput* function is being called by the intelligent controller main Python script in order to step-wisely eliminate the error signal between the desired and the actual magnetic fields, through required electrical current amendment. Table 4.9 briefly describes the functions developed for *PID* Python class.

Function	Description	Parameter(s)	Returns
__init__	Constructs Kp, Ki and Kd gains. Calls *SetUp* function to prepare the PID controller environment.	-	-
SetUp	Initializes the time, term and integral windup guard variables.	-	-
SetKp	Sets the proportional gain.	invar	-
SetKi	Sets the integral gain.	invar	-
SetKd	Sets the derivative gain.	invar	-
GetKp	Gets the proportional gain as a string.	-	str(self.Kp)
GetKi	Gets the integral gain as a string.	-	str(self.Ki)
GetKd	Gets the derivative gain as a string.	-	str(self.Kd)
SetPrevErr	Sets the previous error signal value.	prev_err_sig	-
SetWindup	Sets the windup guard.	windup	-
GenerateOutput	Performs a PID computation and returns a control value based on the pre-set Kp, Ki and Kd gains, elapsed time and the error signal.	error_signal	self.Cp + (self.Ki * self.Ci) + (self.Kd * self.Cd)

Table 4.9: Functions developed for *PID* Python class.

In conclusion, *PID* Python class developed in a way that it can be utilized by any closed-loop system aiming to apply PID (Proportional-Integral-Derivative) process control.

Development of PIDTuning Python Class

As a sub-class of *PID* Python class, *PIDTuning* Python class implemented to handle the tuning of the PID process control. Tuning is essential as PID controller relies on pre-set proportional (Kp), integral (Ki), and derivative (Kd) gains which consecutively affect stability, accuracy, speed and overshooting of the system.

After instantiating the *PID* Python class' object in *PIDTuning* Python class' *__init__* function, another one function developed for the thorough tuning of the system. Especially, *PIDTuning* Python class consists of *Tune* function which uses *SetKp*, *SetKi* and *SetKd* functions of *PID* Python class to set proportional (Kp), integral (Ki) and derivative (Kd) gains, respectively. *Tune* function can then get called by the intelligent controller main Python script for the effective tuning of the PID controller. Figure 4.21 shows the *Tune* function's source code of *PIDTuning* Python class. Table 4.10 outlines the functions produced for *PIDTuning* Python class.

```
18      ''' Tune function setting the Kp, Ki and Kd gains using
19          SetKp, SetKi and SetKd functions of PID class, respectively.
20      #@param: Kp gain
21               Ki gain
22               Kd gain'''
23      def Tune(self, Kp, Ki, Kd):
24          pid.SetKp(Kp)
25          pid.SetKi(Ki)
26          pid.SetKd(Kd)
```

Figure 4.21: *Tune* function's source code of *PIDTuning* Python class.

Function	Description	Parameter(s)
__init__	Initializes the pid global variable used to instantiate the *PID* Python class' object.	-
Tune	Sets the Kp, Ki and Kd gains using *SetKp*, *SetKi* and *SetKd* functions of *PID* Python class, respectively.	Kp, Ki and Kd

Table 4.10: Functions developed for *PIDTuning* Python class.

Finally, *PIDTuning* Python class developed in way that it can be utilized by either manual or auto tuning methods.

4.2.7 Magnet Calibration Feature

Automated magnet calibration feature has been initiated to handle any other magnet than the specified one for this project. On top of that, calibration feature would also ensure accuracy level of the system if the probe (or Hall sensor) is changed. For example, this project uses high sensitivity probe (HSE) and an electromagnet with electrical current limits of +-3 Amps. If for instance the electromagnet remains the same, but the probe (or Hall sensor) is changed to high stability (HST) version, correlation values are expected to be slightly different even with the same electrical current limits of +-3 Amps. Therefore, automated magnet calibration feature would ensure future compatibility of the software with any other magnet, as well as, any other probe (or Hall sensor) version. The following discussion covers the development of *MagnetCalibration* Python class.

Development of MagnetCalibration Python Class

Custom-initiated *MagnetCalibration* Python class is responsible for calibrating the system for a new magnet based on the set electrical current limits and precision level. For this work, increment/decrement steps of the electrical current are pre-set to 0.25 Amps.

After the preparation of magnet calibration through *__init__* function, another ten functions have been developed for the adequate calibration procedure. In particular, *MagnetCalibration* Python class consists of *SetPositiveCurrentLimit*, *SetNegativeCurrentLimit*, *GetPositiveCurrentLimit*, *GetNegativeCurrentLimit*, *GetA_Ascending*, *GetB_Ascending*, *GetA_Descending*, *GetB_Descending*, *ascending_calibration* and *descending_calibration* functions.

For more accurate automated magnet calibration procedure, both ascending and descending changes of the electrical current (Amps) are being monitored along with the corresponding magnetic field (Oe) values at the specified precision level (0.25 Amps).

Both, *ascending_calibration* and *descending_calibration* functions of *MagnetCalibration* Python class are based on equations [68]:

$$a = \frac{\Sigma Y \, \Sigma X^2 - \Sigma X \, \Sigma XY}{n * \Sigma X^2 - (\Sigma X)^2}, \tag{4.5}$$

and

$$b = \frac{n * \Sigma XY - \Sigma X \, \Sigma Y}{n * \Sigma X^2 - (\Sigma X)^2}, \tag{4.6}$$

where X is the electrical current (Amps), Y is the magnetic field (Oe), n is the number of trialed electrical current values, a is either a_ascending or a_descending and b is either b_ascending or b_descending.

Furthermore, all four *a_ascending*, *b_ascending*, *a_descending* and *b_descending* results are gathered by the intelligent controller main Python script to form the ”automated” conversion equation, using *GetA_Ascending*,
GetB_Ascending, *GetA_Descending* and *GetB_Descending* functions, respectively.

Conversion equation which calculates the required electrical current amendment based on the PID controlled output (in the form of feedback signal) is:

$$Electrical\_Current\_Change = \frac{PID\_Output - a}{b}, \tag{4.7}$$

More details on how magnet calibration feature and electrical current conversion equation are being used in the intelligent controller main Python script are followed in section 4.2.9. Table 4.11 briefly describes the functions of *MagnetCalibration* Python class.

Function	Description	Parameter(s)	Returns
__init__	Prepares the magnet calibration environment. By default, system is calibrated for a magnet with electrical current limits of +-3 Amps.	-	-
SetPositiveCurrentLimit	Sets the positive electrical current limit.	invar	-
SetNegativeCurrentLimit	Sets the negative electrical current limit.	invar	-
GetPositiveCurrentLimit	Gets the positive electrical current limit.	-	self.PosCurrLim
GetNegativeCurrentLimit	Gets the negative electrical current limit.	-	self.NegCurrLim
GetA_Ascending	Gets the a_ascending variable resulted from calibration procedure.	-	self.a_ascending
GetB_Ascending	Gets the b_ascending variable resulted from calibration procedure.	-	self.b_ascending
GetA_Descending	Gets the a_descending variable resulted from calibration procedure.	-	self.a_descending
GetB_Descending	Gets the b_descending variable resulted from calibration procedure.	-	self.b_descending

ascending_calibration	Calibrates the system for a new magnet based on the set electrical current limits and precision level (0.25 Amps). Calibration follows ascending changes of electrical current.	-	-
descending_calibration	Calibrates the system for a new magnet based on the set electrical current limits and precision level (0.25 Amps). Calibration follows descending changes of electrical current.	-	-

Table 4.11: Functions developed for *MagnetCalibration* Python class.

4.2.8 User Interface (UI)

Usability of the proposed intelligent controller based on a Raspberry Pi system had to be considered closely, due to the user interaction with the program. Therefore, Human-computer interaction (HCI) properties of content awareness, layout, aesthetics and user experience considered carefully during the initiation of the terminal-based UI (User Interface) [69]. The following discussion covers the development of UI Python class. As time permitted, and for enhancing the overall user experience in the system, a simple GUI (Graphical User Interface) has also been developed which is discussed in section 4.2.10.

Development of UI Python Class

UI (User Interface) Python class was developed to assist the intelligent controller main Python script which will be discussed further in section 4.2.9. At first instance, *UI*

Python class is utilizing *__init__* function to set the environment and configure variables that will be used by other functions at later stage.

Furthermore, *UserInterface* function of *UI* Python class is structured in a way that is giving four different options to the end-user. Namely, user can either run the program, calibrate for different magnet, enable/disable the plotting feature or exit the program. Selected option of the user is monitored for the coherent execution of the program.

It is important that *UserInterface* function includes appropriate type and value exceptions for validating the user input. For instance, if the end-user selects to run the program, then he/she is prompted to input the desire magnetic field. Therefore, if the desired magnetic field entered by the user exceeds the bounded field limits (based on magnet calibration) or if the input format is wrong (e.g. string entered), an exception is raised, terminating the program and informing the user about the error. More detailed discussion of how errors are handled by the *UI* Python class is followed in section 5.2. Figure 4.22 shows a visual representation of the terminal-based UI main window.

Figure 4.22: Visual representation of the terminal-based UI main window.

Other functions of *UI* Python class include *GetOptionNumber*, *GetDesiredMagneticField*, *GetCurrentLimit* and *GetPlotFeatureOnOff*. In particular, these functions are used by the intelligent controller main Python script for the coherent interaction of the end-user with the terminal based UI. Table 4.12 briefly describes all the functions produced for *UI* Python class.

Function	**Description**	**Returns**
__init__	Constructing variables that are being used from other functions of *UI* class.	-
UserInterface	Configures the terminal-based UI. Four different options for the user - 0: Exit Program, 1: Run Program, 2: Calibrate for different magnet (default +-3 Amps) and 3: Enable/Disable plotting feature (default enabled).	self.option
GetOptionNumber	Gets the option number selected by the user.	self.option
GetDesiredMagneticField	Gets the desired magnetic field value entered by the user.	float(self.desire_magnetic_field)
GetCurrentLimit	Gets the current limit value entered by the user for magnet calibration.	float(self.current_limit)
GetPlotFeatureOnOff	Gets the selection of the user for enabling/disabling plotting option.	self.plot_feature_onoff

Table 4.12: Functions developed for *UI* Python class.

4.2.9 Final Software Integration

For an effective usage of the produced Python classes, a main script was required. The following discussion includes the final software integration through *IntelligentController_UI.py* main Python script.

Development of IntelligentController_UI Main Python Script

The following steps describe the final software integration and how the program handles different use-case scenarios through *IntelligentController_UI.py* main Python script:

1. Imported *LakeShore425_Gauss*, *KepcoBOP*, *MagnetCalibration*, *SerialInterface*, *GpibInterface*, *PID*, *PIDTuning* and *UI* custom-initiated Python classes. External math, matplotlib, time and datetime software libraries have also been included for doing the required math conversions, plotting the control system trials, managing the delays between different processes and showing real-time updates to the end-user, accordingly.

2. Defined variables and instantiated the objects of custom-initiated Python classes. Instantiated objects named as gaussmeter, power_supplier, magnet_calibration, ser, gpib, pid_controller, tuning and user_interface. These objects were used to call the appropriate functions from their Python classes.

3. Prepared the environment for communicating with the Lake Shore 425 gaussmeter using both *c_FieldRange* and *c_FieldUnits* functions of the *LakeShore425_Gauss* Python class in conjunction with the *serialWriteOnly* function of the *SerialInterface* Python class.

4. Prepared the environment for communicating with the Kepco BOP power supply using *c_SetRemoteCommunication*, *c_SetVoltageLevel*, *c_OperatingMode*, *c_EDOutput* and *SetCurrentLevel* functions of the *KepcoBOP* Python class in conjunction with the *gpibWriteOnly* function of the *GpibInterface* Python class. Although at later stage of the software only electrical current has to be controlled for the modification of the magnetic field, voltage has to be set at the beginning of the program as well for a thorough functionality.

5. Tuned the system with the pre-defined proportional (Kp), integral (Ki) and derivative (Kd) gains using *Tune* function of the *PIDTuning* Python class, preparing the program to enter into the PID process loop.

6. In a closed-loop context, at first instance, *UserInterface* function of the *UI* Python class creates the terminal-bsaed UI and asks the user to select an option. Four different options are available to the end-user, including 0: Exit Program, 1: Run Program, 2: Calibrate for different magnet (default +-3 Amps) and 3: Enable/Disable plotting feature (default enabled).

7. Gets the option number selected by the user, using the *GetOptionNumber* function of the *UI* Python class.

8. If the user selects option 0, program terminates.

9. If the user selects option 1, program outcomes the actual magnetic field with the current timestamp using the *serialFullCommunication*, *q_FieldReading* and *now* functions of the *SerialInterface*, *LakeShore425_Gauss* and *datetime* libraries, respectively. Then, using *raw_input* function, program prompts the user to enter the desired magnetic field. As the *raw_input* function saves the desired magnetic field input as a string, conversion to float value is required. After gathering the desired magnetic field input from the user, the following sub-steps describe the PID control loop procedure:

   (a) Actual magnetic field is read again. Comparison between the desire and actual magnetic fields is essential in order to find the initial error signal.

   (b) If the desire magnetic field input is equals to the actual magnetic field reading, error signal is set to 0. Otherwise, if the desire magnetic field input is less or greater than the actual magnetic field reading, supplied electrical current for the actual magnetic field reading is calculated based on either descending or ascending A and B values of the magnet calibration.

   (c) Then, PID process is taking place using *GenerateOutput* function of the *PID* Python class.

   (d) After that, required electrical current for the PID feedback signal is calculated based on either descending or ascending A and B values of the magnet calibration. It is important that both descending and ascending changes are monitored for more accurate and stable PID feedback signal.

   (e) Adding both supplied electrical current for the actual magnetic field reading and the required electrical current for the PID feedback signal, results with the desire electrical current amendment. It is important to make clear

that both supplied electrical current for the actual magnetic field reading and the required electrical current for the PID feedback signal are needed because PID control process is a continuous procedure, resulting with a gradual (and not the overall) change (either descending or ascending) towards to the desired output.

(f) Then, the desire electrical current is converted into a scientific form (using *str*) and remotely transmitted to the Kepco BOP power supply using *gpibWriteOnly* and *c_SetCurrentLevel* functions of the *GpibInterface* and *KepcoBOP* Python classes, accordingly.

(g) Steps 9(a), 9(b), 9(c), 9(d), 9(e) and 9(f) are executed for a pre-set number of iterations. For this system, 50 iterations are optimum.

(h) After the PID control loop iterations are over, and thus, the error signal is eliminated, actual magnetic field output is printed to the terminal-based UI along with the current timestamp, available to the end-user.

(i) Graphical representation of the PID control loop run is also made available to the user, if the plotting feature of the program is enabled.

(j) After the PID control loop procedure is over, program is keep checking, until the next desired magnetic field input is being entered by the end-user.

10. If the user selects option 2, program is prompting the user to input the electrical current limits for the new magnet. Based on the set electrical current limits, system is automatically calibrated for a different magnet for both ascending and descending changes of electrical current.

11. If the user selects option 3, program is prompting the user to either enable or disable the plotting feature by entering 1 or 0, accordingly.

Figure 4.23 shows a section of the *IntelligentController_UI.py* Python script source code which is handling the main process of the system, including the storage of the desired magnetic field input, the reading of the actual magnetic field, the calculation of the error signal, the computation of the PID output in the form of feedback signal and the remote amendment of the electrical current as desired.

Different unit and integrated software tests for the intelligent controller based on a Raspberry Pi are discussed in sections 5.2 and 5.3, respectively.

```
        ''' if user selects to run the control program. '''
        if (option == RunOption):
            desire_magnetic_field_reading = user_interface.GetDesiredMagneticField()
            # start the timer (milliseconds)
            start_time = int(round(time.time() * 1000))
            ''' run 50 trials for reaching the desire magnetic field based on the feedback signal. '''
            for i in range(1,NUM_OF_TRIAL_ITERATIONS):
                ''' get the actual magnetic field using the hall sensor. '''
                actual_magnetic_field_reading = float(ser.serialFullCommunication(gaussmeter.q_FieldReading()))
                ''' calculate the error signal (desire_magnetic_field_reading - actual_magnetic_field_reading). '''
                error_signal = float(desire_magnetic_field_reading) - float(actual_magnetic_field_reading)
                ''' Split into ascending and descending changes for more accuracy and stability. '''
                if (desire_magnetic_field_reading < actual_magnetic_field_reading): # descending change
                    ''' calculate the actual electrical current based on set magnet calibration. '''
                    actual_electrical_current = ((actual_magnetic_field_reading - magnet_calibration.GetA_Descending())/magnet_calibration.GetB_Descending())
                    ''' save the PID output as feedback_signal. '''
                    feedback_signal = pid_controller.GenerateOutput(error_signal)
                    ''' calculate the feedback electrical current based on set magnet calibration. '''
                    feedback_electrical_current = ((feedback_signal - magnet_calibration.GetA_Descending())/magnet_calibration.GetB_Descending())
                elif(desire_magnetic_field_reading > actual_magnetic_field_reading): # ascending change
                    ''' calculate the actual electrical current based on set magnet calibration. '''
                    actual_electrical_current = ((actual_magnetic_field_reading - magnet_calibration.GetA_Ascending())/magnet_calibration.GetB_Ascending())
                    ''' save the PID output as feedback_signal. '''
                    feedback_signal = pid_controller.GenerateOutput(error_signal)
                    ''' calculate the feedback electrical current based on set magnet calibration. '''
                    feedback_electrical_current = ((feedback_signal - magnet_calibration.GetA_Ascending())/magnet_calibration.GetB_Ascending())
                else:
                    ''' if there is no difference between desire_magnetic_field_reading and actual_magnetic_field_reading. '''
                    error_signal = 0.0

                ''' add the actual_electrical_current and the feedback_electrical_current. '''
                desire_electrical_current = actual_electrical_current + feedback_electrical_current
                ''' round the desire_electrical_current using the float_round function. '''
                rounded_desire_electrical_current = float_round(desire_electrical_current,DECIMAL_PLACES,ceil)
                ''' convert the float desire_electrical_current value to string - ready to communicate with the power supplier. '''
                string_desire_electrical_current = str(rounded_desire_electrical_current) + "E0"
                ''' set the current level using gpib. '''
                gpib.gpibWriteOnly(power_supplier.c_SetCurrentLevel(string_desire_electrical_current))
                ''' give some time before the next run. '''
                time.sleep(PID_DELAY)

                # end the timer (milliseconds)
                elapsed_time = (int(round(time.time() * 1000)) - start_time)
```

Figure 4.23: A section of the *IntelligentController_UI.py* Python script source code which is handling the main process of the system, including the storage of the desired magnetic field input, the reading of the actual magnetic field, the calculation of the error signal, the computation of the PID output in the form of feedback signal and the remote amendment of the electrical current as desired.

4.2.10 Graphical User Interface (GUI)

As time permitted, a simple GUI (Graphical User Interface) has been implemented using Tkinter Python library, enhancing the overall usability of the system. The following discussion covers the development of GUI through *IntelligentController_GUI.py* main Python script.

Development of IntelligentController_GUI Main Python Script

At the software level, functionality principle of the *IntelligentController_GUI.py* script is the same with the *IntelligentController_UI.py* script. However, in order to design and implement the GUI, software in *IntelligentController_GUI.py* had to be revised

to serve two different Python classes. Namely, *IntelligentController_GUI.py* includes *MainWindow* and *PopUpWindow* Python classes.

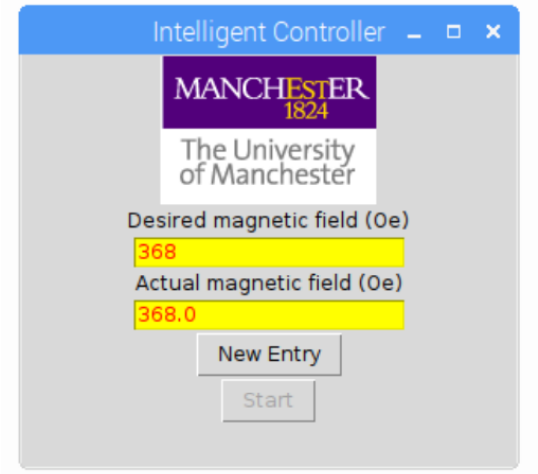
In particular, *MainWindow* Python class of *IntelligentController_GUI.py* script is responsible for the main process of the system, including front-end functionalities of starting the control procedure using Start button and calling the pop-up window using New Entry button. Back-end functionalities of *MainWindow* Python class include, but not limited to:

- Gathering the desired magnetic field input.
- Reading the actual magnetic field.
- Calculating the error signal.
- Computing the PID output in the form of feedback signal.
- Modifying the electrical current as desired.

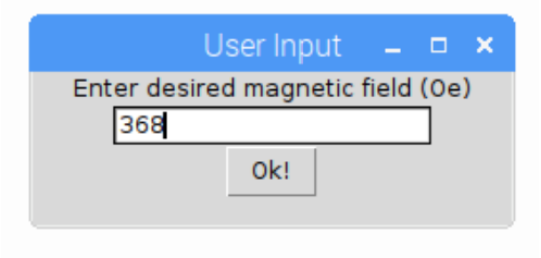
Above back-end functionalities of the *MainWindow* Python class are executed in a closed-loop manner, conforming with the proposed system's fundamental principle. *PopUpWindow* Python class of *IntelligentController_GUI.py* script is handling the insertion of desired magnetic field input from the end-user. Figure 4.24 shows both main and pop-up windows of GUI, implemented for the purposes of intelligent controller based on a Raspberry Pi system.

As a storyboard procedure, when GUI is first initiated it shows only the main window with the Start button disabled in order to prompt the user to enter the desired magnetic field using the New Entry button. When the New Entry button is pressed, pop-up window appears and prompts the user to input the desired magnetic field. After entering the desired magnetic field value and pressing "Ok!" button, the pop-up window disappears, leaving only the GUI main window with the Start button now enabled. Hence, user can either run the program using Start button or enter a different desired magnetic field using the New Entry button once again.

Finally, yet importantly, both desired and actual magnetic field entry boxes are disabled at all times in order to prevent the user from modifying any values. User can exit the GUI at any time by pressing the cross (x) button at the top-right of the main window.



(a) Main window of the GUI.



(b) Pop-up window of the GUI.

Figure 4.24: Main and pop-up windows of the GUI, implemented for the purposes of the intelligent controller based on a Raspberry Pi system.

Chapter 5

TESTING, RESULTS, EVALUATION AND OPTIMIZATION

After the design and implementation of the intelligent controller based on a Raspberry Pi system, testing and evaluation were crucial in order to remark, and thus, guarantee the adequate behaviour of the automated control solution. This chapter discusses different hardware and software unit tests that have been conducted, together with the integrated system tests. Testing and evaluation plans (see section 4.1.4) initiated during the design phase of the project have been followed for a thorough experimentation of the system. Results of the integrated system tests have been analyzed, compared and evaluated, accordingly. After the determination of the fit-for-purpose tuning configuration for the system, speed optimization has also been undertaken.

Furthermore, every control system is evaluated based on its SASO (Stability, Accuracy, Speed and Overshooting) performance properties outcome. For this project, SASO performance indices have been prioritized as:

1. Stability.
2. Absence of overshooting (or undershooting).
3. Accuracy (along with repeatability).
4. Speed (or settling time).

The following discussion covers the hardware and software unit testing (see sections 5.1 and 5.2), together with the integrated system testing (see section 5.3), results (see section 5.4) and speed optimization (see section 5.5).

5.1 Hardware Unit Verification/Testing

For early experimentation purposes, hardware unit verification/testing for both LakeShore425 gaussmeter and Kepco BOP power supply had to be conducted. The following discussion includes the gaussmeter (see section 5.1.1) and power supply (see section 5.1.2) verification/testing.

5.1.1 Gaussmeter Verification/Testing

Prior to moving to the software implementation of the system, Lake Shore 425 gaussmeter's initial communication was verified/tested. With the Lake Shore 425 gaussmeter being connected to the host PC through USB, verification/testing was conducted by serially transmitting different message strings using Lake Shore communication tool (see section 3.4.4). Queries prompted through Lake Shore communication tool include "*IDN?\ r\ n", "UNIT?\ r\ n" and "RDGFIELD?\ r\ n" for getting the identity of the instrument, reading the set magnetic field unit and gathering the actual magnetic field reading, respectively. Figure 5.1 shows the prompted message strings (queries) along with the testing results obtained through Lake Shore communication tool, verifying that the Lake Shore 425 gaussmeter is responsive.

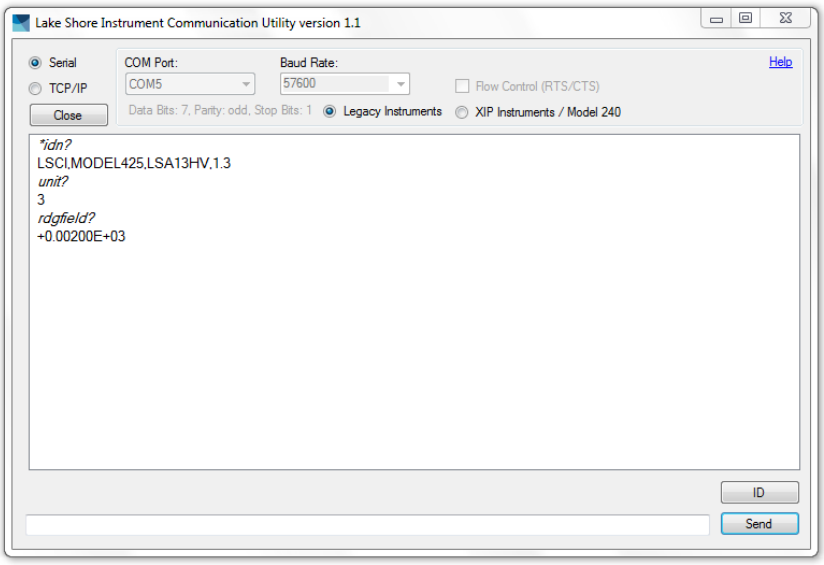


Figure 5.1: Hardware unit verification/test of Lake Shore 425 gaussmeter using Lake Shore communication tool. Lake Shore 425 gaussmeter is responsive to all prompted queries.

5.1.2 Power Supply Verification/Testing

For early experimentation purposes, Kepco BOP power supply's initial communication had to be verified/tested as well, prior to moving to the software implementation of the system. With the Kepco BOP power supply being connected to the host PC using National Instruments GPIB-USB-HS control device, verification/testing was conducted by prompting different SCPI message strings (commands/queries) through NI VISA interactive control tool (see section 3.4.5).

SCPI message strings transmitted through the NI VISA interactive control tool include "CURR 3EO\ n" command and "CURR?\ n" query for setting the electrical current to 3 Amps and reading the supplied electrical current level, accordingly. Figure 5.2 illustrates the prompted SCPI command which is setting the electrical current level to 3 Amps, while Figure 5.3 shows the prompted SCPI query (and its response) which is reading the supplied electrical current level straight after the command is being sent.

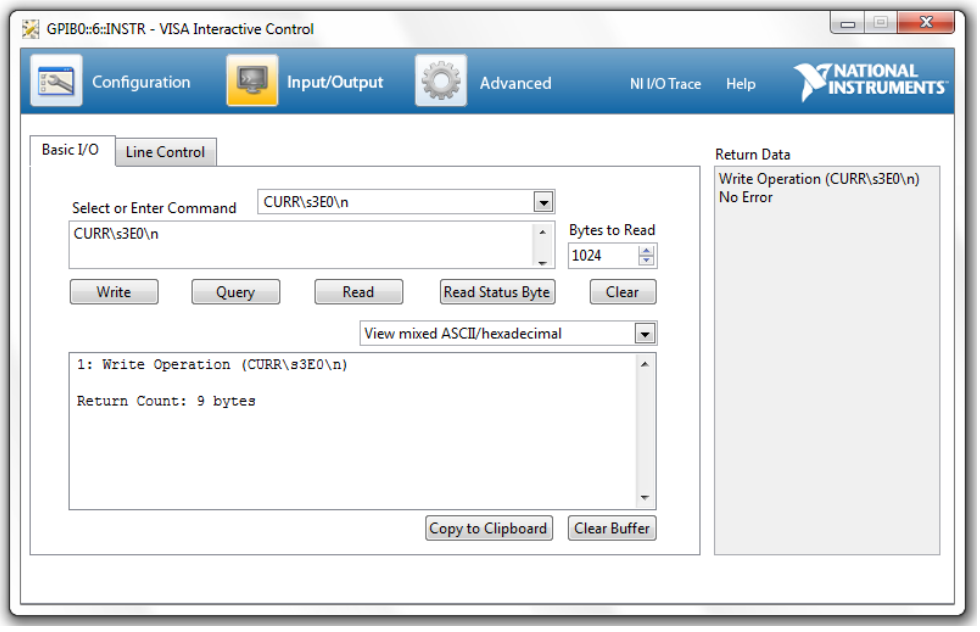


Figure 5.2: Hardware unit verification/test of Kepco BOP power supply using NI VISA interactive control tool. Sets the electrical current to 3 Amps.

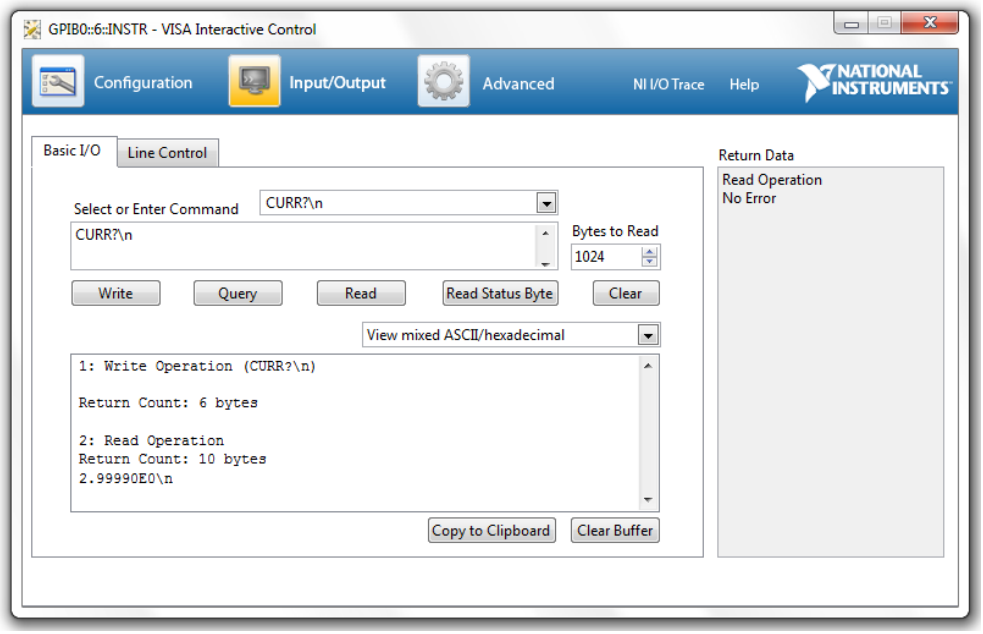


Figure 5.3: Hardware unit verification/test of Kepco BOP power supply using NI VISA interactive control tool. Reads the supplied electrical current level straight after the command is being sent. Kepco BOP power supply is responsive at optimum level (supplied electrical current reading is ∼2.99 Amps).

Testing results gathered from NI VISA interactive control tool are verifying that Kepco BOP power supply is responsive at optimum level (supplied electrical current reading is ∼2.99 Amps). Consecutively, above hardware unit test verified that the built-in BIT 4886 digital interface card of the Kepco BOP power supply is functioning properly.

5.2 Software Unit Testing

In the context of a software engineering project, software unit testing had to be considered closely as un-tested scenarios could result the system to become ”uncontrolled”. After conducting the hardware unit verification/testing (see section 5.1) and implementing the required software (see section 4.2), it was time to investigate different software use-case scenarios. Following the testing and evaluation plans (see section 4.1.4) introduced during the design phase of the project, software unit tests of validating the desired magnetic field input format and any other functionalities of the system have been conducted.

Software unit tests for the intelligent controller based on a Raspberry Pi system

have been conducted using the terminal-based UI (User Interface). In particular, four different options are available to the end-user upon terminal-based UI's instantiation:

- **0**: Exit Program.
- **1**: Run Program.
- **2**: Calibrate for different magnet (default +-3 Amps).
- **3**: Enable/Disable plotting feature (default enabled).

The following discussion covers software unit tests undertaken for the initial creation of the terminal-based UI (see section 5.2.1) together with the exit (see section 5.2.2), run (see section 5.2.3), magnet calibration (see section 5.2.4) and plot (see section 5.2.5) features of the terminal-based UI.

5.2.1 Instantiating the User Interface (UI)

First software unit test concerns with the instantiation of the terminal-based UI. If the user tries to start the program and either Lake Shore 425 gaussmeter or Kepco BOP power supply are switched off, UI quits and informs the user with the appropriate exception message. Figure 5.4 shows how the program reacts when either Lake Shore 425 gaussmeter or Kepco BOP power supply are switched off and user tries to start the terminal-based UI.

```
pi@raspberrypi:~ $ cd Desktop/Project_Directory/
pi@raspberrypi:~/Desktop/Project_Directory $ sudo python IntelligentController_UI.py
Traceback (most recent call last):
  File "IntelligentController_UI.py", line 162, in <module>
    control_loop()
  File "IntelligentController_UI.py", line 88, in control_loop
    setup()
  File "IntelligentController_UI.py", line 66, in setup
    ser = serialInterface.SerialInterface()
  File "/home/pi/Desktop/Project_Directory/Interfaces/serialInterface.py", line 22, in __init__
    dsrdtr = False
  File "/usr/lib/python2.7/dist-packages/serial/serialutil.py", line 236, in __init__
    self.open()
  File "/usr/lib/python2.7/dist-packages/serial/serialposix.py", line 268, in open
    raise SerialException(msg.errno, "could not open port {}: {}".format(self._port, msg))
serial.serialutil.SerialException: [Errno 2] could not open port /dev/ttyUSB0: [Errno 2] No such file or directory: '/dev/ttyUSB0'
pi@raspberrypi:~/Desktop/Project_Directory $ sudo python IntelligentController_UI.py
Traceback (most recent call last):
  File "IntelligentController_UI.py", line 162, in <module>
    control_loop()
  File "IntelligentController_UI.py", line 88, in control_loop
    setup()
  File "IntelligentController_UI.py", line 72, in setup
    magnet_calibration = MagnetCalibration.MagnetCalibration()
  File "/home/pi/Desktop/Project_Directory/Calibration/MagnetCalibration.py", line 52, in __init__
    gpib.gpibWriteOnly(power_supplier.c_SetRemoteCommunication(1))
  File "/home/pi/Desktop/Project_Directory/Interfaces/gpibInterface.py", line 40, in gpibWriteOnly
    " (cannot write/read).")
Exception: Gpib communication error (cannot write/read).
pi@raspberrypi:~/Desktop/Project_Directory $
```

Figure 5.4: Software unit test - if the user tries to start the program and either Lake Shore 425 gaussmeter or Kepco BOP power supply are switched off, UI quits and informs the user with the appropriate exception message.

5.2.2 Exit Feature of User Interface (UI)

In a user interface, exit feature must be made available to the end-user at all times, following the fundamental usability principles. In order to validate the exit feature of the proposed terminal-based UI, option 0 (Exit Program) has been selected. Figure 5.5 demonstrates how the exit feature is handled by the terminal-based UI of the intelligent controller system.

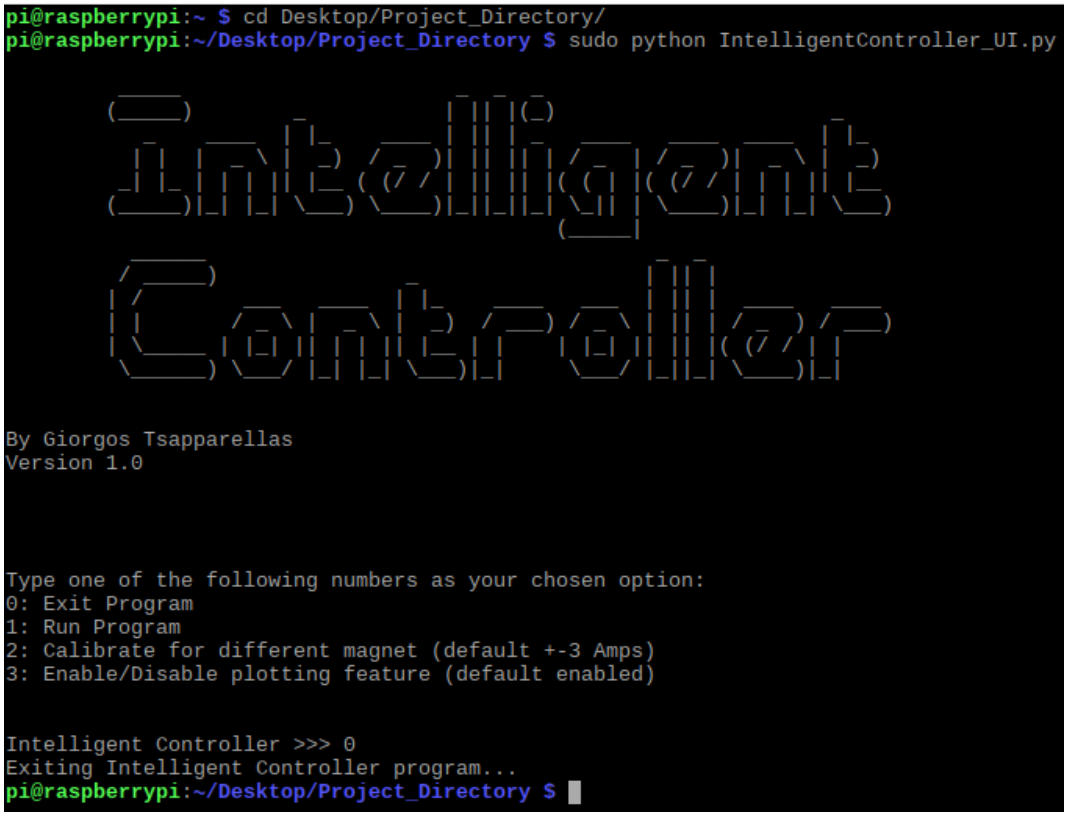


Figure 5.5: Software unit test - program terminates if the user selects the 0 (Exit Program) option.

5.2.3 Run Feature of User Interface (UI)

As a core requirement of the project, user must be able to enter the desired magnetic field input through the termnal-based UI. However, end-user must provide the desired magnetic field input as numeric value. String format is not accepted. Depending on the specified magnet's capabilities, user is also bounded to the maximum magnetic field input that he/she can entered. Therefore, electrical current flowing to the coils of the electromagnet will never exceed its set limitations.

In order to validate the acceptable format and boundaries for the desired magnetic field input, two different software unit tests have been conducted. At first instance, Figure 5.6 shows how irregular format of the desired magnetic field input is handled by the program using appropriate type exception. Secondly, Figure 5.7 illustrates how an out-of-boundaries desired magnetic field input is handled by the program using the appropriate value exception. In both cases, after the exception has been thrown to the user, program quits to guarantee that the system will not be driven out-of-control.

Finally, a software unit test has been conducted to investigate the successful run of the intelligent controller program through the terminal-based UI. Figure 5.8 demonstrates how the intelligent controller based on a Raspberry Pi system handles a successful run of the program, after the user has input the desired magnetic field in the correct format and within the set boundary limits.

Figure 5.6: Software unit test - irregular format of the desired magnetic field input.

Figure 5.7: Software unit test - desired magnetic field input out-of-boundaries.

Figure 5.8: Software unit test - successful run of the intelligent controller program.

5.2.4 Magnet Calibration Feature of User Interface (UI)

Magnet calibration feature has been added at the final stages of the software implementation. This feature is giving the opportunity to the end-user to calibrate the system for a brand new magnet, based on the set electrical current limit. Similarly to the *run* feature of the terminal-based UI, magnet calibration feature only accepts electrical current limit input as a numeric value. String format is not tolerable.

In order to validate the acceptable format for the electrical current limit input, a software unit test has been conducted. Figure 5.9 shows how irregular format of the electrical current limit input is managed by the program using the appropriate type exception. Another software unit test has been conducted to observe how the program

reacts on successful magnet calibration. Figure 5.10 illustrates how the intelligent controller based on a Raspberry Pi system handles a successful magnet calibration, after the end-user has entered the electrical current limit in the correct format.

Figure 5.9: Software unit test - irregular format of the electrical current limit input (for magnet calibration purposes).

Figure 5.10: Software unit test - successful magnet calibration.

5.2.5 Plot Feature of User Interface (UI)

Plot feature was useful during the development phase in order to investigate different tuning (manual or auto) configurations. However, at user level, plot feature might not be required. End-user must be able to ”opt-out” from the plot feature at any time using the terminal-based UI. In particular, plot feature of the program can be enabled by entering number 1 or disabled by prompting number 0. By default, plot feature of the intelligent controller based on a Raspberry Pi system is enabled (1). In order to validate the acceptable format for enable (1)-disable (0) input, three different software unit tests have been conducted.

Regards to the first software unit test, Figure 5.11 demonstrates how irregular format of the enable (1)-disable (0) input is handled by the program using the appropriate value exception. In addition, another two software unit tests have been conducted to remark how the program acknowledges a successful enablement or disablement of plot

feature. Figures 5.12 and 5.13 show how the program handles both successful enablement and disablement of plot feature, after the end-user has entered the enable (1) and disable (0) values in the correct format, respectively.

Figure 5.11: Software unit test - irregular format of the enable (1)-disable (0) input (for plot feature purposes).

Figure 5.12: Software unit test - successful enablement of plot feature.

Figure 5.13: Software unit test - successful disablement of plot feature.

5.3 Integrated System Testing

After the approval of the hardware and software unit tests (see sections 5.1 and 5.2), integrated system testing has been conducted. The integrated system testing of the intelligent controller based on a Raspberry Pi followed the testing and evaluation plans (see section 4.1.4) introduced during the design phase of the project. Overall system functionality and usability were tested through the integration of the intelligent controller into a standalone instrument, as a part of multi-component system.

Furthermore, the integrated system testing has been conducted through the custom-initiated UI (User Interface) and focused on controlling the magnetic field of an electromagnet, including the remote modification of the electrical current supplied to the coils of the electromagnet, after the desired magnetic field has been entered by the user

through a host PC. Tuning was an essential part of the integrated system testing as the proposed PID controller relies on pre-set proportional (Kp), integral (Ki) and derivative (Kd) gains which consecutively affect the stability, accuracy, speed and overshooting of the system. Therefore, different tuning methods (manual and auto) have been observed and compared in order to identify the best gains (Kp, Ki and Kd) configuration for the proposed PID controller by prompting different desired magnetic fields through the terminal-based UI. After the determination of the fit-for-purpose tuning configuration for the system, speed optimization has been undertaken.

PID control process is a continuous procedure, maintaining the system with the providence of gradual feedback signal (positive or negative) towards to the desired output. Therefore, a pre-set number of iterations for the PID control loop had to be defined for testing purposes. For this system, 50 iterations were optimum for the run of the control loop in order to allow the PID controller to step-wisely reach the desired magnetic field prompted by the end-user. Plot feature of the terminal-based UI was useful during the integrated system testing in order to show the gradual changes introduced by the PID controller towards to the desired magnetic field set-point.

Moreover, in order to allow the control of the intelligent controller based on a Raspberry Pi during the integrated system testing, without the utilization of an external monitor and directly from the host PC, VNC Viewer tool (see sections 3.4.3 and 4.2.1) has been used. It is important that VNC Viewer third-party tool has no latency, and therefore, did not affect the performance of the system. Absence of VNC Viewer's latency was verified by first connecting the Raspberry Pi to an external monitor and conduct some initial integrated system tests.

The following discussion includes manual (see section 5.3.1) and auto (see section 5.3.2) tuning of the intelligent controller based on a Raspberry Pi, including Ziegler-Nichols and Tyreus-Luyben methods.

5.3.1 Manual Tuning

For the initial integrated system testing, different manual tuning configurations have been observed to select the most optimum Kp, Ki and Kd gains for the proportional (P), integral (I) and derivative (D) modes, accordingly. Individual effects of the proportional (Kp), integral (Ki) and derivative (Kd) gains considered closely for a thorough manual tuning of the system (see section 2.4.7). Full manual tuning tests log for the intelligent controller based on a Raspberry Pi system can be found in Appendix G-Table G.1.

It is important that ”anti-windup” feature of the software has been intentionally excluded from some manual tuning tests in order to investigate any possible signs of either overshooting or undershooting in the system.

The following figures show the manual tuning tests conducted for the purposes of the intelligent controller based on a Raspberry Pi. P, I and D values of the following graphs referred to as the proportional (Kp), integral (Ki) and derivative (Kd) gains.

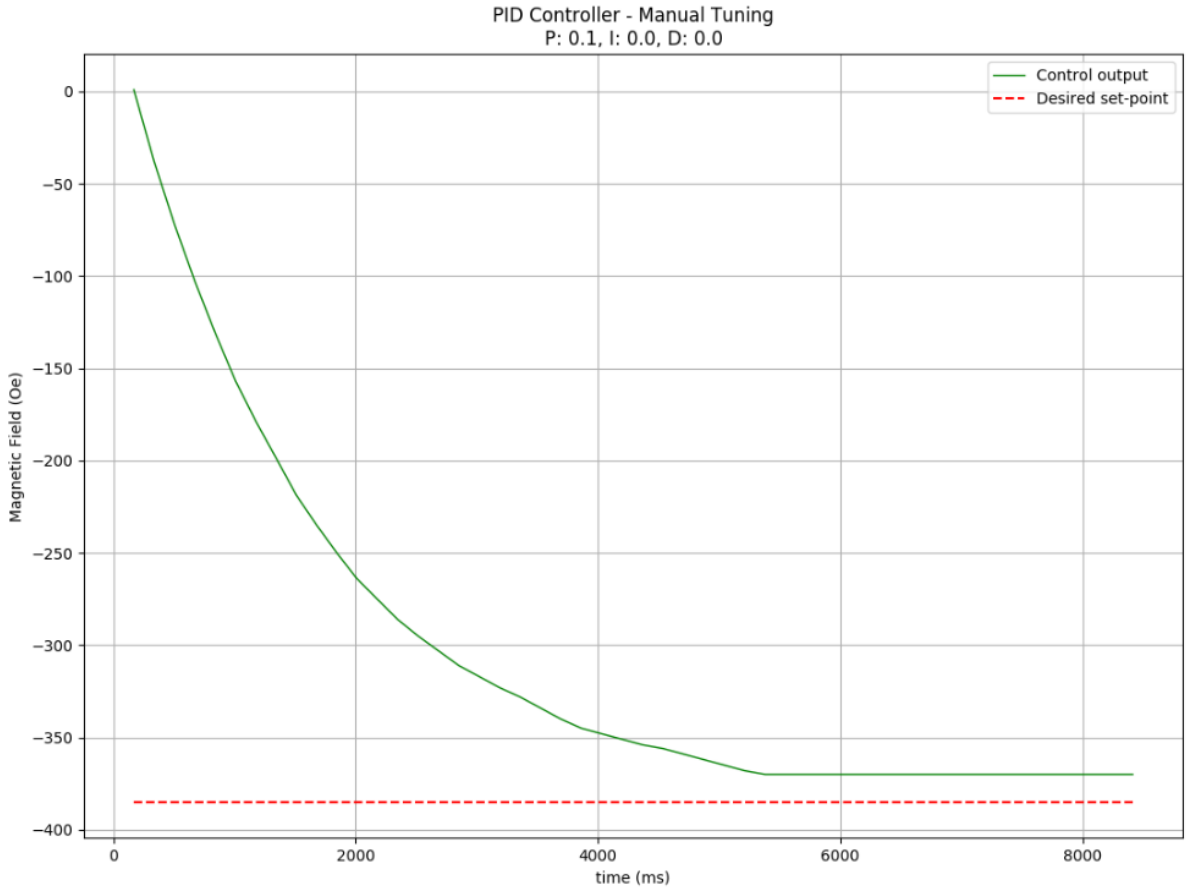


Figure 5.14: Manual tuning of the intelligent controller based on a Raspberry Pi. P-only type of controller with pre-set gain Kp = 0.1. Desired magnetic field is -385 Oe and starting magnetic field is 0 Oe. Initial error is -385 Oe, reached magnetic field is -370 Oe and settling time is 5700 ms. This tuning configuration outcomes steady-state error (or ”droop”).

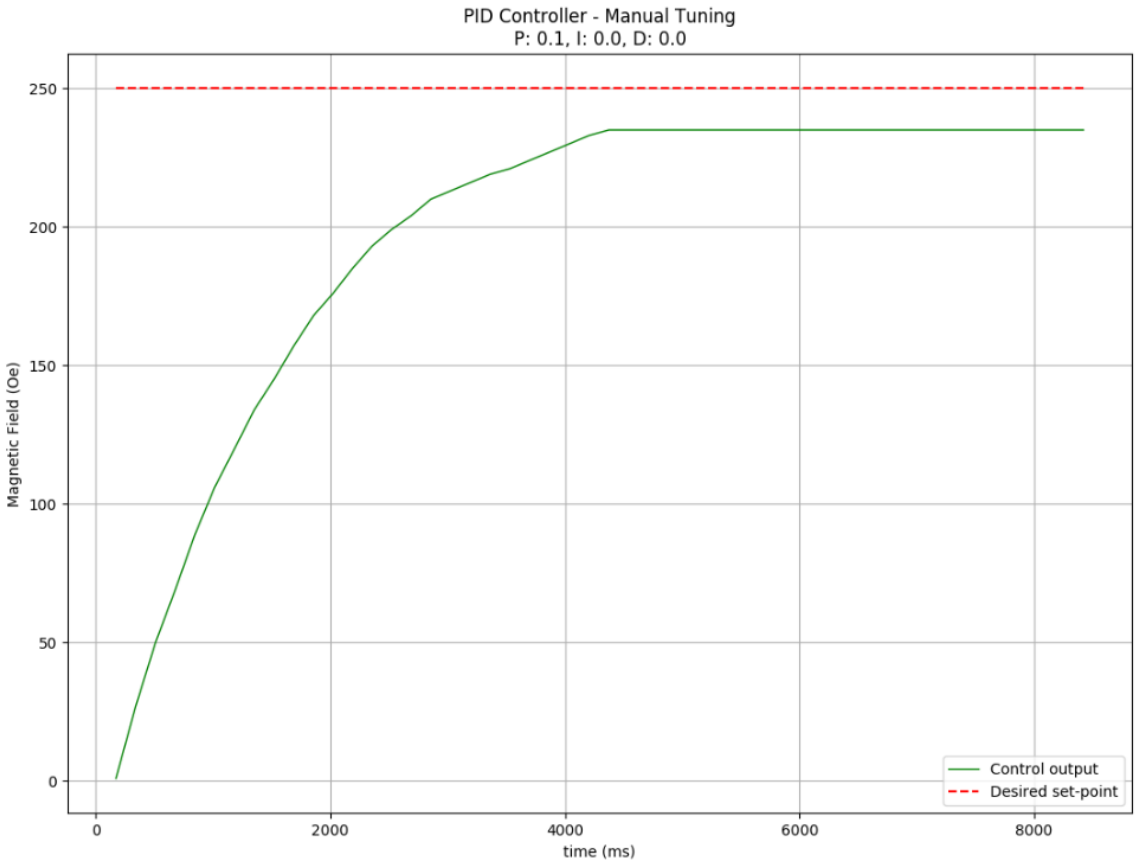


Figure 5.15: Manual tuning of the intelligent controller based on a Raspberry Pi. P-only type of controller with pre-set gain Kp = 0.1. Desired magnetic field is 250 Oe and starting magnetic field is 0 Oe. Initial error is 250 Oe, reached magnetic field is 235 Oe and settling time is 4200 ms. This tuning configuration outcomes steady-state error (or ”droop”).

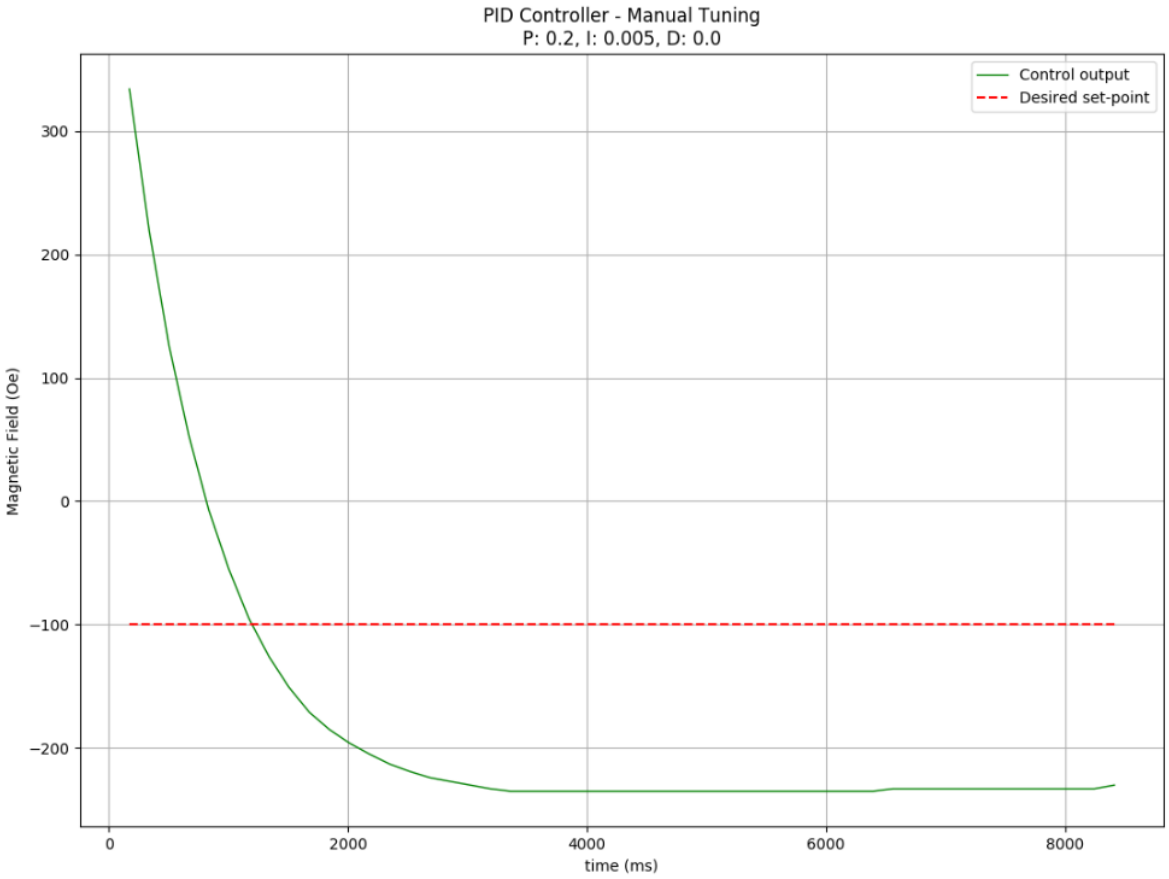


Figure 5.16: Manual tuning of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.2 and Ki = 0.005. Desired magnetic field is -100 Oe and starting magnetic field is 334 Oe. Initial error is -434 Oe, reached magnetic field is -225 Oe and settling time is 3500 ms. This tuning configuration outcomes undershooting problem.

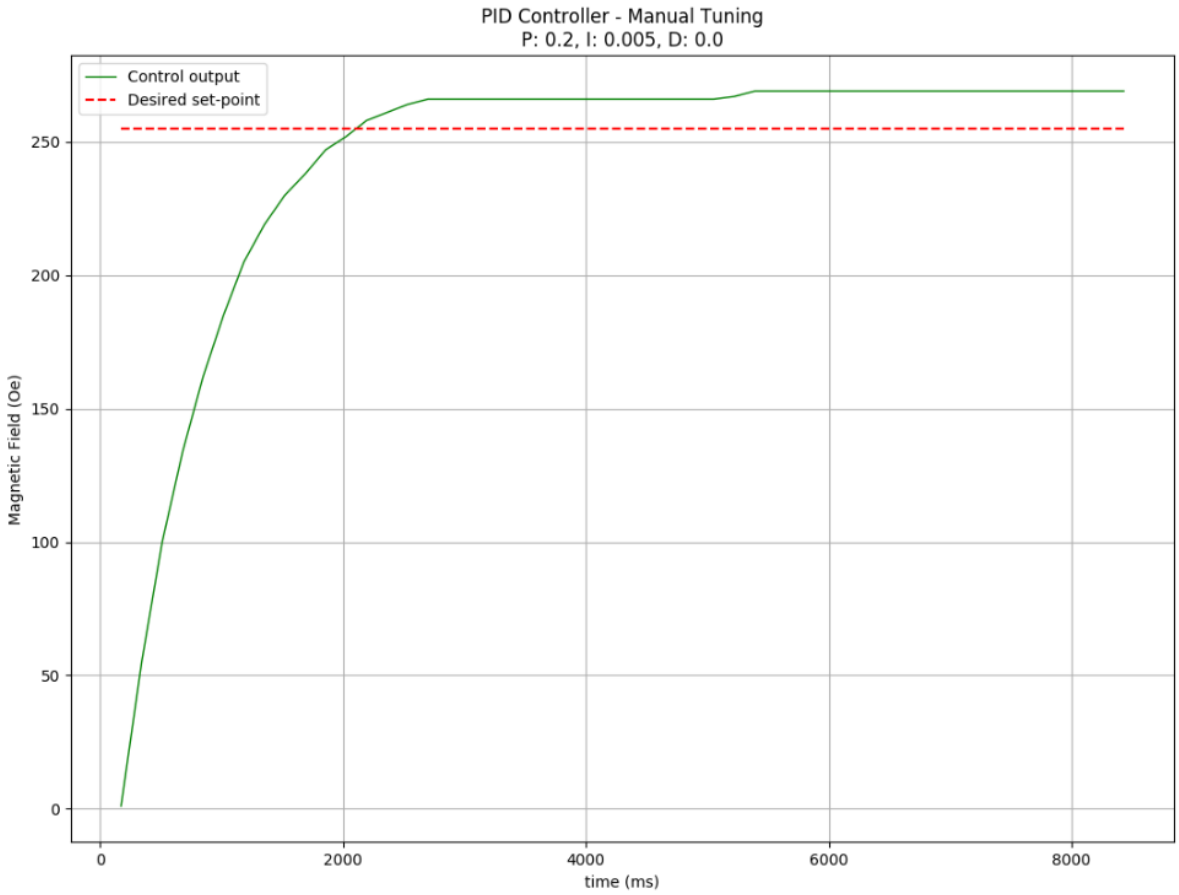


Figure 5.17: Manual tuning of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.2 and Ki = 0.005. Desired magnetic field is 255 Oe and starting magnetic field is 0 Oe. Initial error is 255 Oe, reached magnetic field is 270 Oe and settling time is 5700 ms. This tuning configuration outcomes overshooting problem.

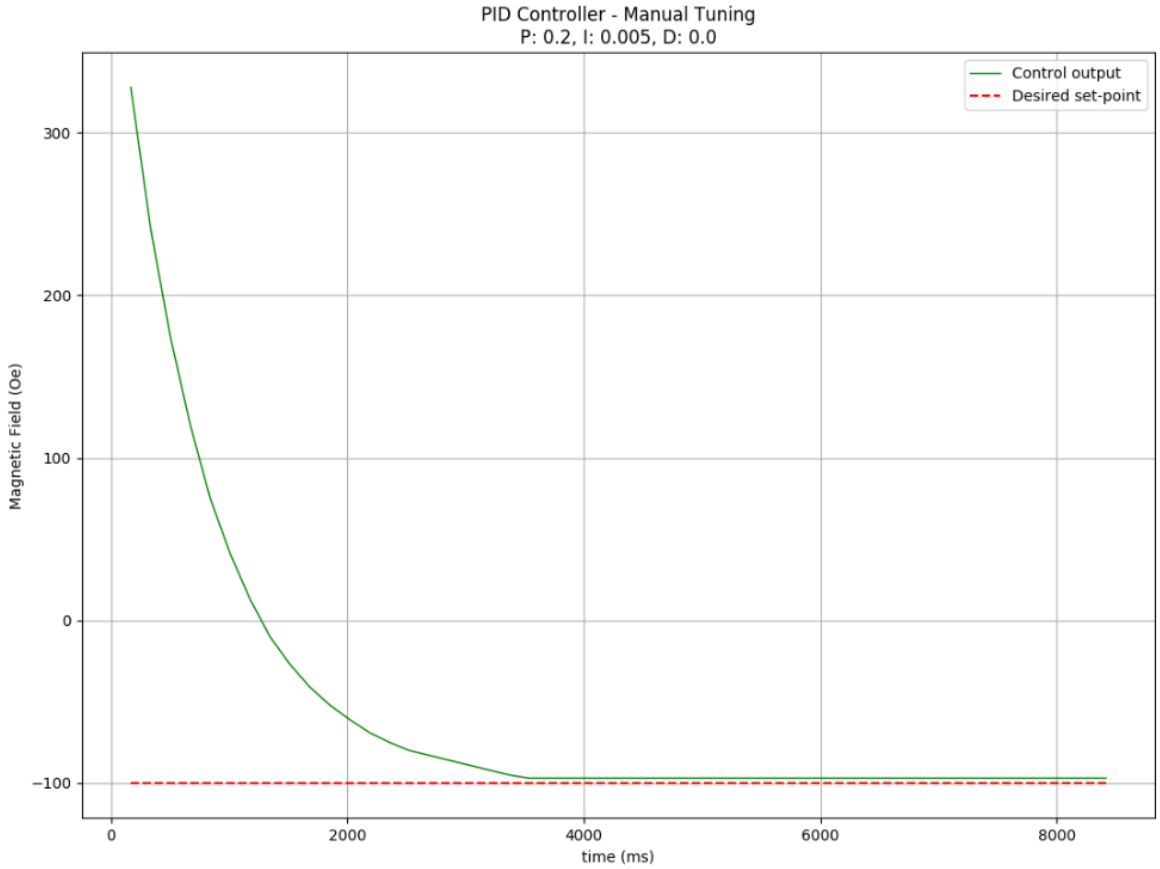


Figure 5.18: Manual tuning of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.2 and Ki = 0.005. Desired magnetic field is -100 Oe and starting magnetic field is 328 Oe. Initial error is -428 Oe, reached magnetic field is -97 Oe and settling time is 3800 ms. This tuning configuration eliminates undershooting problem with ”anti-windup” feature of the software, but still outcomes steady-state error (or ”droop”).

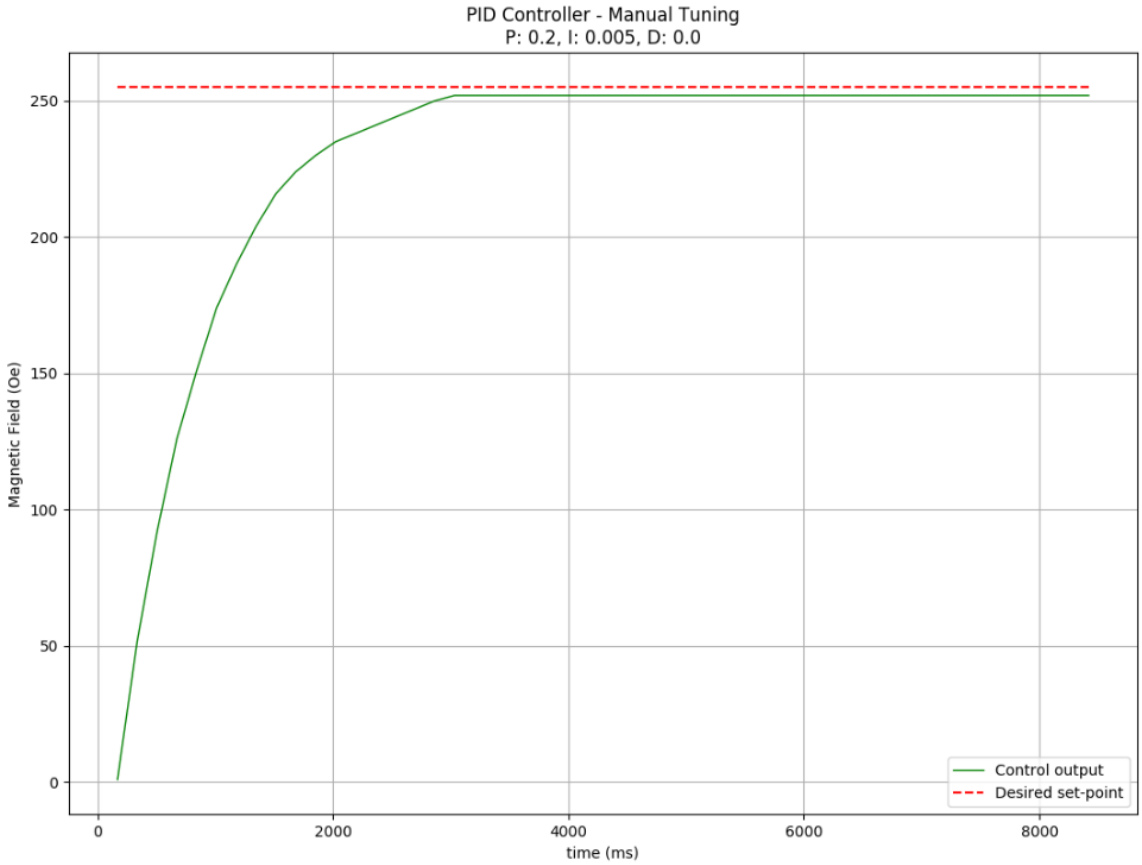


Figure 5.19: Manual tuning of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.2 and Ki = 0.005. Desired magnetic field is 255 Oe and starting magnetic field is 0 Oe. Initial error is 255 Oe, reached magnetic field is 252 Oe and settling time is 3000 ms. This tuning configuration eliminates overshooting problem with ”anti-windup” feature of the software, but still outcomes steady-state error (or ”droop”).

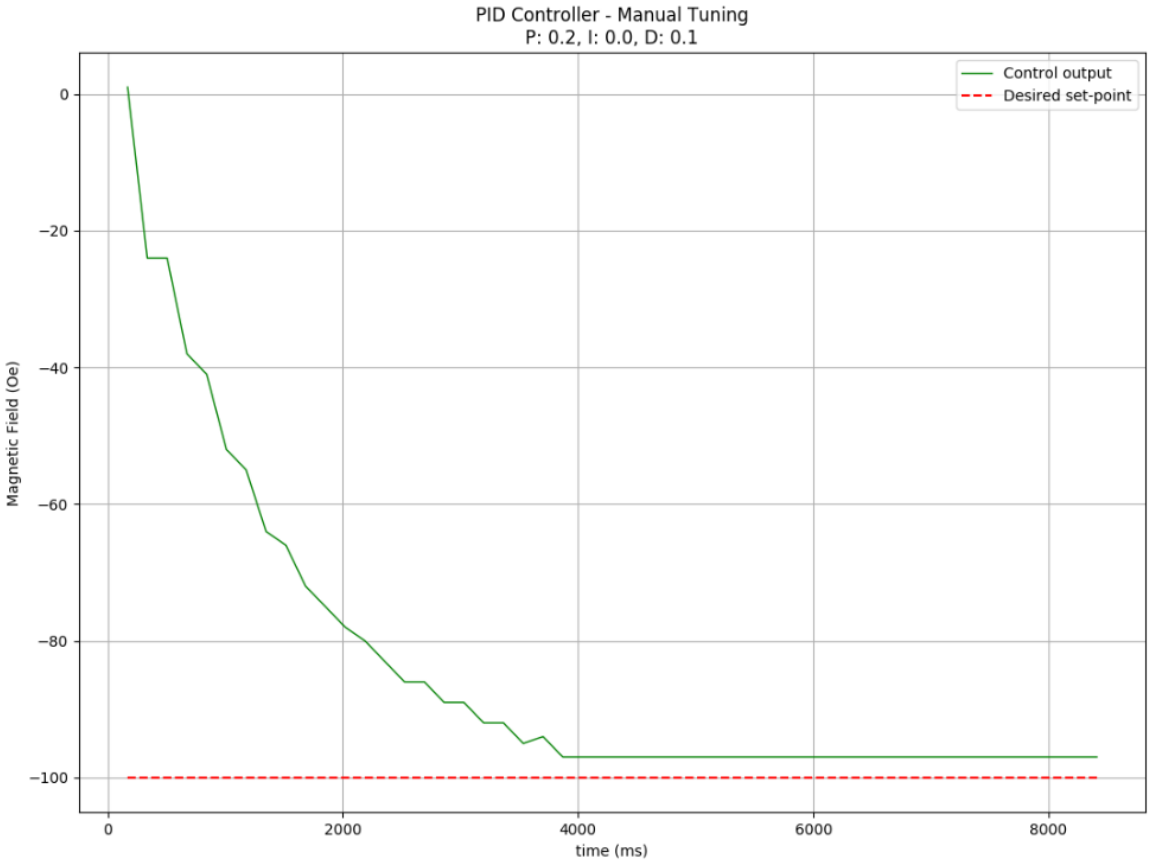


Figure 5.20: Manual tuning of the intelligent controller based on a Raspberry Pi. PD type of controller with pre-set gains Kp = 0.2 and Kd = 0.1. Desired magnetic field is -100 Oe and starting magnetic field is 0 Oe. Initial error is -100 Oe, reached magnetic field is -97 Oe and settling time is 4000 ms. This tuning configuration outcomes steady-state error (or ”droop”). Signal is noisy due to derivative (D) term.

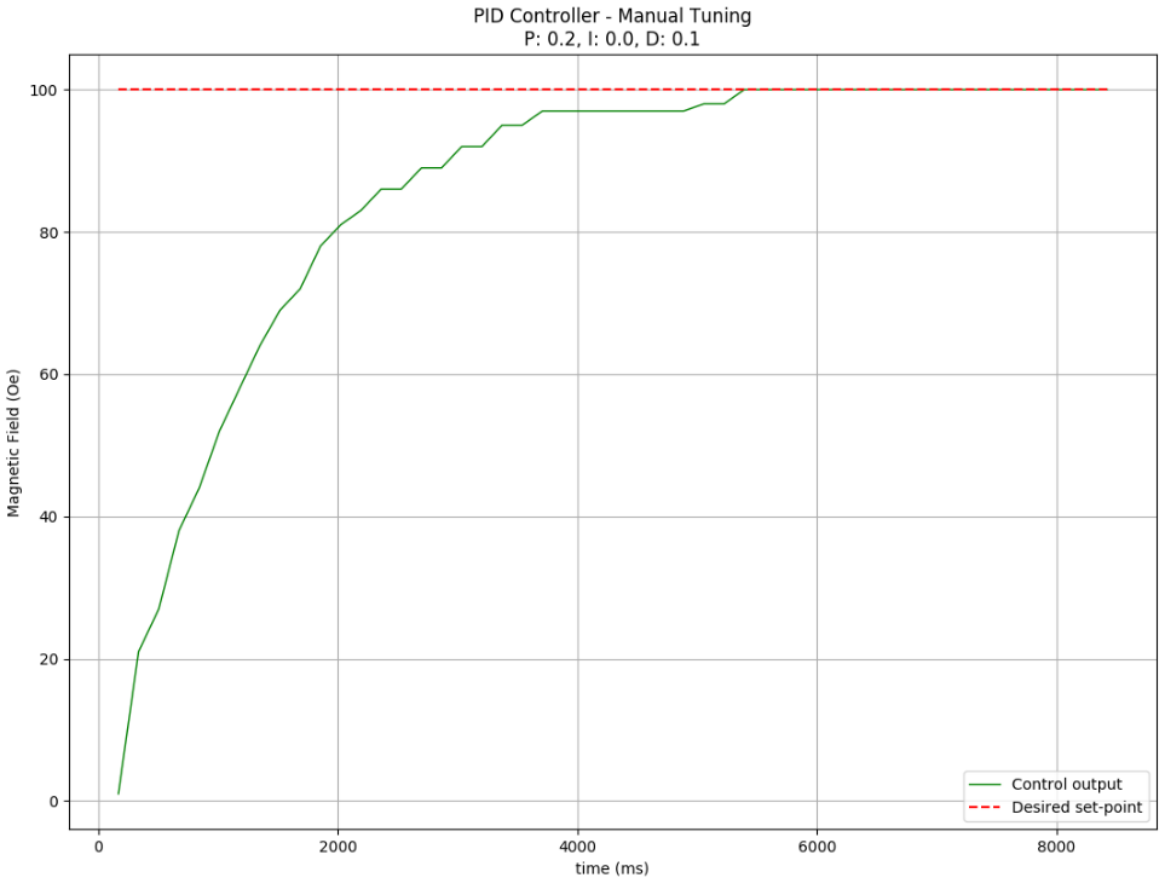


Figure 5.21: Manual tuning of the intelligent controller based on a Raspberry Pi. PD type of controller with pre-set gains Kp = 0.2 and Kd = 0.1. Desired magnetic field is 100 Oe and starting magnetic field is 0 Oe. Initial error is 100 Oe, reached magnetic field is 100 Oe and settling time is 5800 ms. This tuning configuration reaches the desired magnetic field, but signal is noisy due to derivative (D) term and takes 5800 ms to settle.

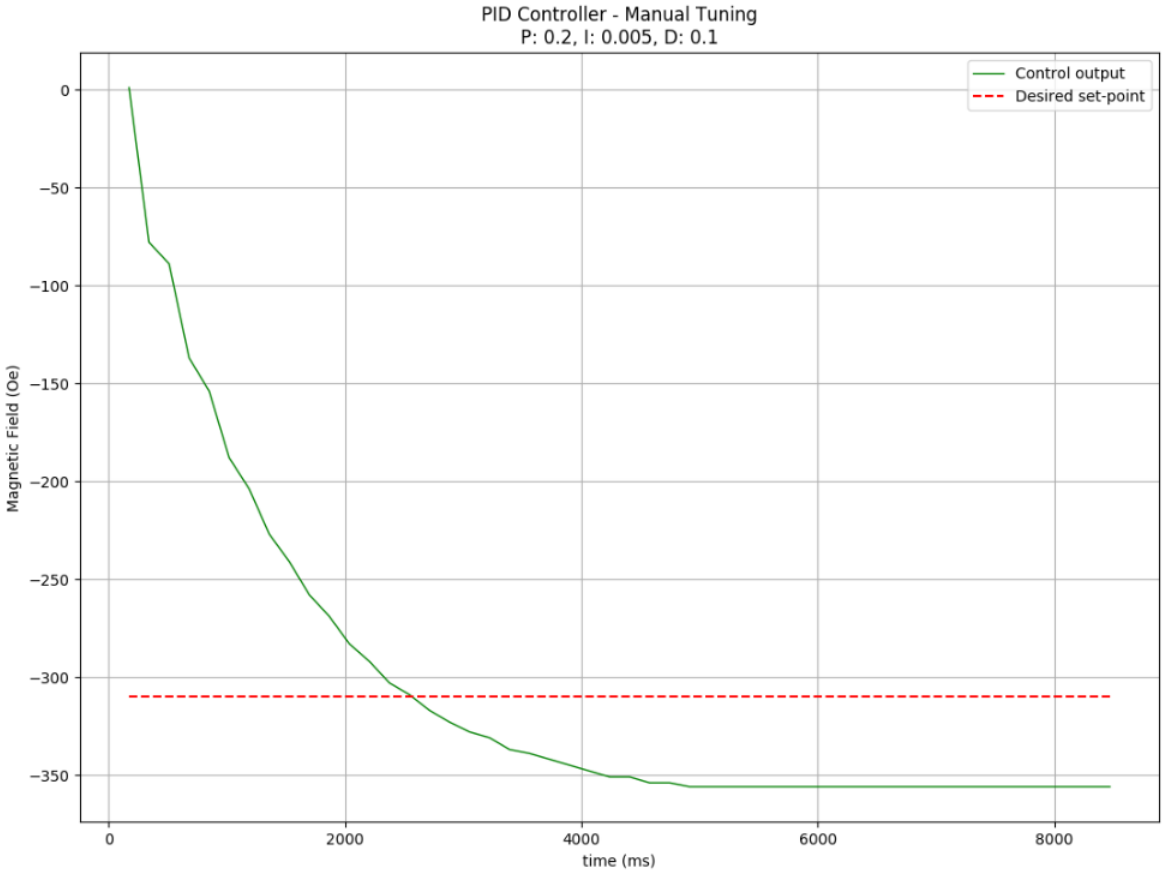


Figure 5.22: Manual tuning of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.2, Ki = 0.005 and Kd = 0.1. Desired magnetic field is -310 Oe and starting magnetic field is 0 Oe. Initial error is -310 Oe, reached magnetic field is -356 Oe and settling time is 5000 ms. This tuning configuration outcomes undershooting problem and signal is noisy due to derivative (D) term.

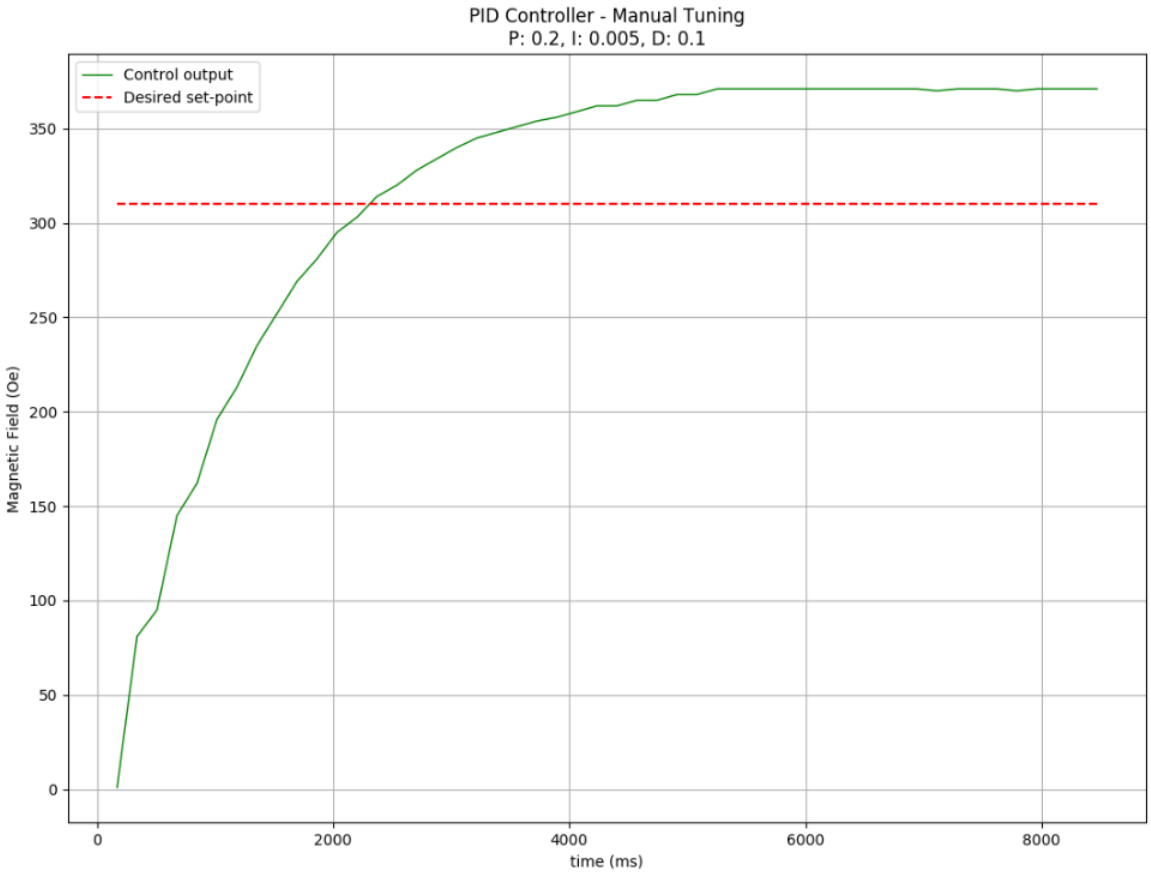


Figure 5.23: Manual tuning of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.2, Ki = 0.005 and Kd = 0.1. Desired magnetic field is 310 Oe and starting magnetic field is 0 Oe. Initial error is 310 Oe, reached magnetic field is 371 Oe and settling time is 5000 ms. This tuning configuration outcomes overshooting problem and signal is noisy due to derivative (D) term.

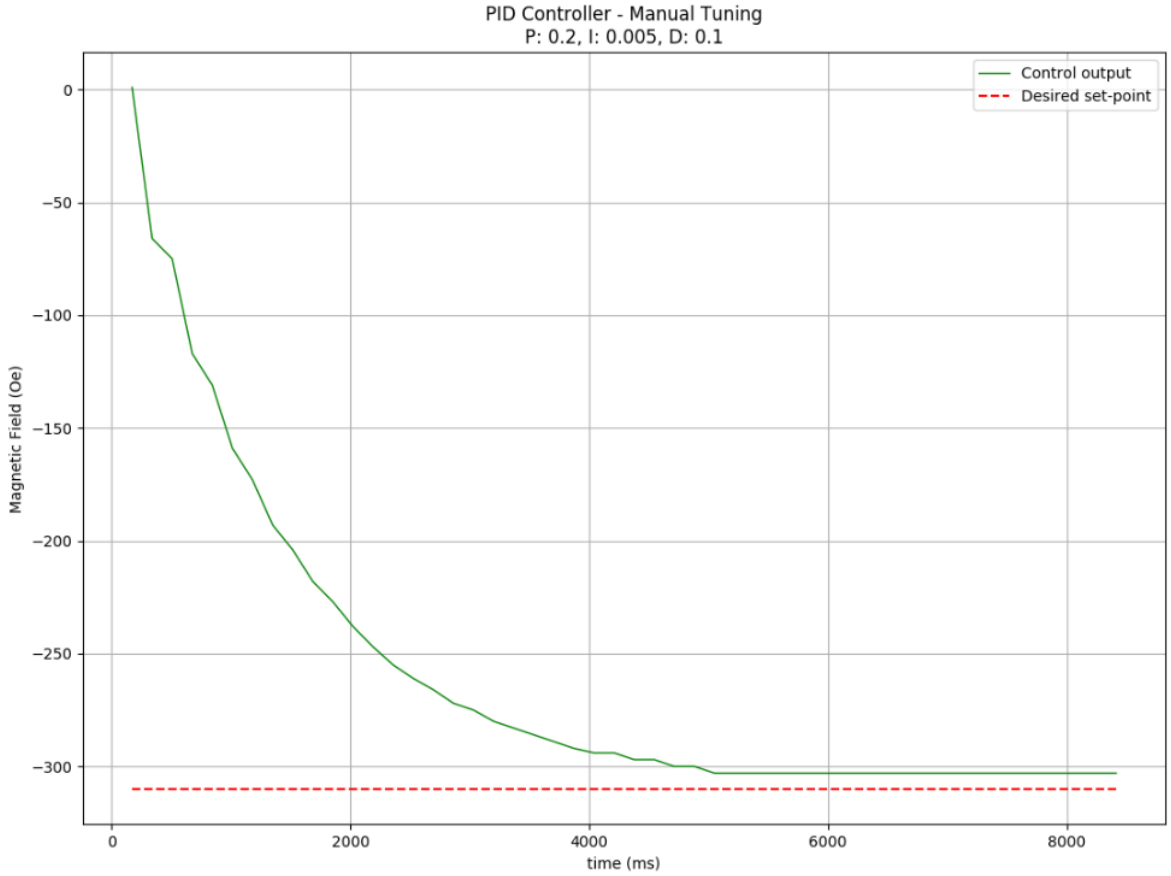


Figure 5.24: Manual tuning of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.2, Ki = 0.005 and Kd = 0.1. Desired magnetic field is -310 Oe and starting magnetic field is 0 Oe. Initial error is -310 Oe, reached magnetic field is -303 Oe and settling time is 5000 ms. This tuning configuration eliminates undershooting problem with ”anti-windup” feature of the software, but still outcomes steady-state error (or ”droop”). Signal is noisy due to derivative (D) term.

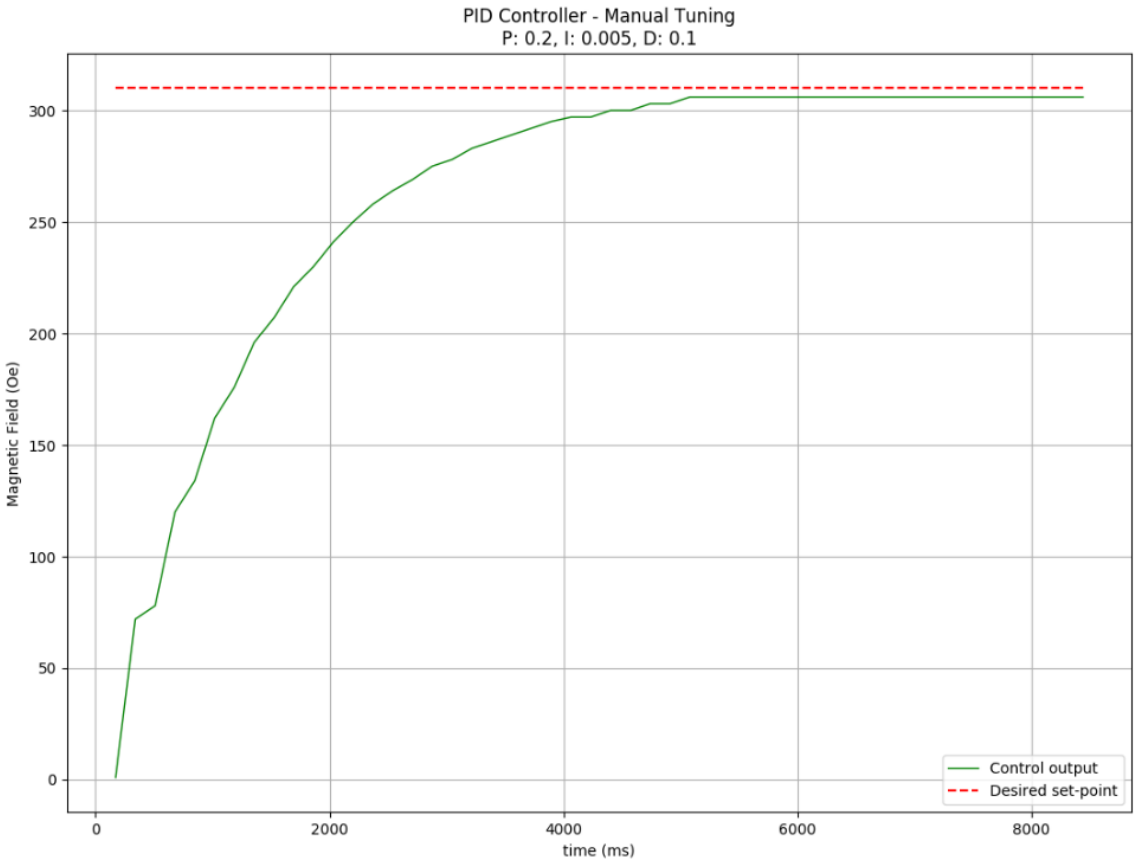


Figure 5.25: Manual tuning of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.2, Ki = 0.005 and Kd = 0.1. Desired magnetic field is 310 Oe and starting magnetic field is 0 Oe. Initial error is 310 Oe, reached magnetic field is 306 Oe and settling time is 5000 ms. This tuning configuration eliminates overshooting problem with ”anti-windup” feature of the software, but still outcomes steady-state error (or ”droop”). Signal is noisy due to derivative (D) term.

5.3.2 Auto Tuning

Manually tuning the proportional (Kp), integral (Ki) and derivative (Kd) gains of P, I and D control modes provides flexibility in the control procedure and gives some initial understanding of the PID process. However, as it is based on trial and error procedure, manual tuning is time-consuming and requires experience. Therefore, auto tuning methods of Ziegler-Nichols (or ultimate cycle) and Tyreus-Luyben have been used to further tune the proposed intelligent controller based on a Raspberry Pi. Results of Ziegler-Nichols (or ultimate cycle) and Tyreus-Luyben auto tuning methods

have been evaluated and compared, resulting with the most optimum tuning configuration for the system which was then used to undertake speed optimization (see section 5.5). It is important that "anti-windup" feature of the software has been included in all the tests of auto tuning (Nichols-Ziegler and Tyreus-Luyben methods), as well as, speed optimization in order to prevent any possible signs of either overshooting or undershooting in the system.

Ziegler-Nichols Method

In order to apply Ziegler-Nichols auto tuning method to the intelligent controller based on a Raspberry Pi, certain steps had to be performed before [1]:

1. Set the integral (Ki) and derivative (Kd) gains to zero.
2. Set the proportional (Kp) gain to a low value and introduce a set-point (or reference input).
3. Increase the proportional (Kp) gain until the system goes to sustained periodic oscillation.
4. Record the proportional (Kp) gain caused the system to go into sustained periodic oscillation and measure the oscillation period (Tu).
5. These values are referred to as the ultimate gain (Kpu) and the ultimate oscillation period (Tu).

Figure 5.26 shows the intelligent controller based on a Raspberry Pi to go into sustained periodic oscillation when the proportional (Kp) gain is set to 1. Therefore, ultimate gain (Kpu) is equals to 1 as well. On the other hand, ultimate oscillation period (Tu) is referred to as the time captured for two sustained oscillations. In order to measure the ultimate oscillation period (Tu) some calculations required. In fact, as it can be observed from Figure 5.26, the PID control process is running for $\sim$8 seconds by the time it first reaches the desired magnetic field set-point (200 Oe). PID control process is also pre-defined to run for 50 iterations (see section 5.3). Therefore, the following equations resulted with the ultimate oscillation period (Tu) for the intelligent controller based on a Raspberry Pi:

$$Ti = executionTime/numOfIterations \sim= 8/50 \sim= 0.16\,second\ (or\ 160\,ms), \quad (5.1)$$

where i is 1, T1 is the time period of a single oscillation (or cycle), *executionTime* is the overall time taken to run PID control process and *numOfIterations* is the pre-set number of iterations for the executed control loop. Therefore, as the ultimate oscillation period (Tu) is referred to as the time captured for two sustained oscillations (or cycles):

$$Tu = T1 * 2 \sim = 0.32\ second\ (or\ 320\ ms) \tag{5.2}$$

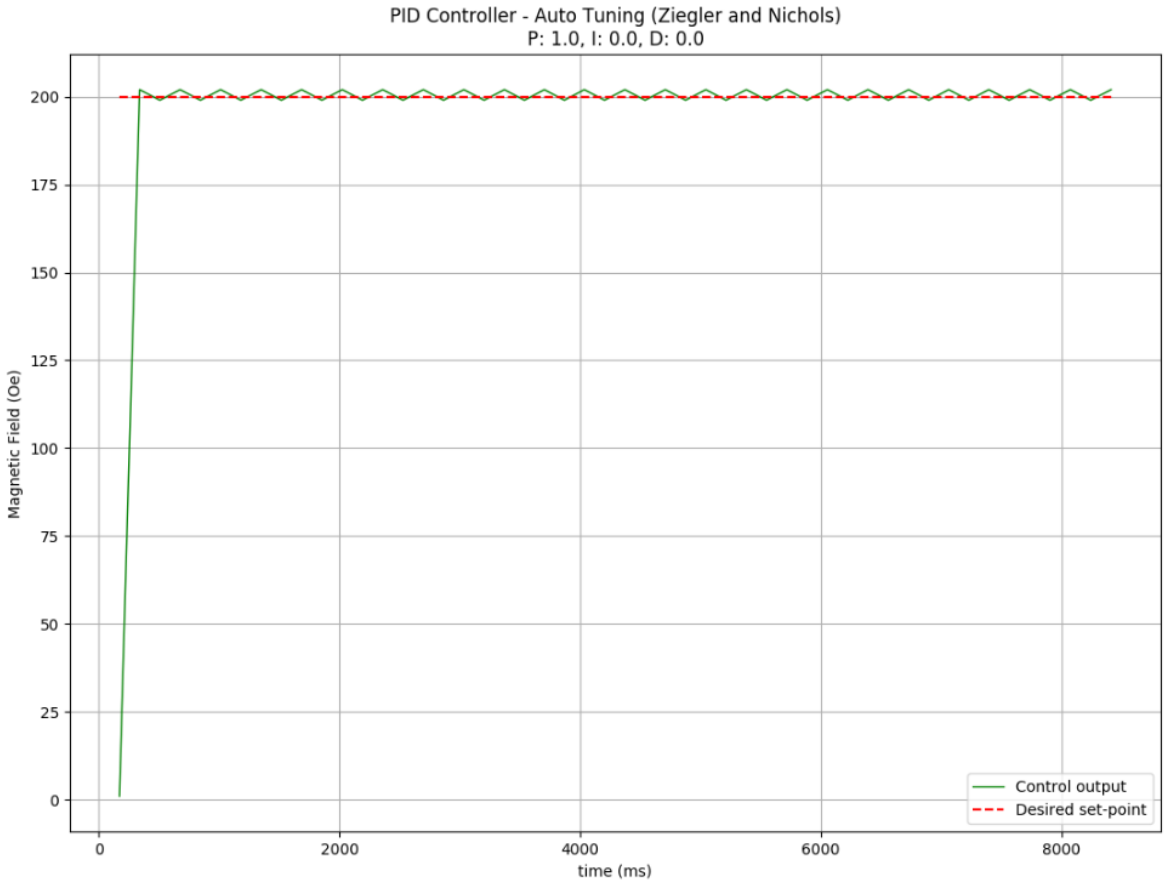


Figure 5.26: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. System goes into sustained periodic oscillation when the proportional (Kp) gain is set to 1.

The extracted ultimate gain (Kpu) and ultimate oscillation period (Tu) values were then used to calculate the proportional (Kp), integral (Ki) and derivative (Kd) gains, using Ziegler-Nichols recommended criteria of Table 2.2. Suggested P-only, PI, and

PID control modes along with their Kp, Ki and Kd gains are P-only (Kp = 0.5, Ki = 0 and Kd = 0), PI (Kp = 0.45, Ki = 0.27 and Kd = 0) and PID (Kp = 0.60, Ki = 0.16 and Kd = 0.04).

The following figures illustrate the Ziegler-Nichols' auto tuning tests conducted for the purposes of the intelligent controller based on a Raspberry Pi using P-only (Kp = 0.5, Ki = 0 and Kd = 0), PI (Kp = 0.45, Ki = 0.27 and Kd = 0) and PID (Kp = 0.60, Ki = 0.16 and Kd = 0.04) tuning configurations. P, I and D values of the following graphs referred to as the proportional (Kp), integral (Ki) and derivative (Kd) gains. Full Ziegler-Nichols auto tuning tests log for the intelligent controller based on a Raspberry Pi system can be found in Appendix H-Table H.1.

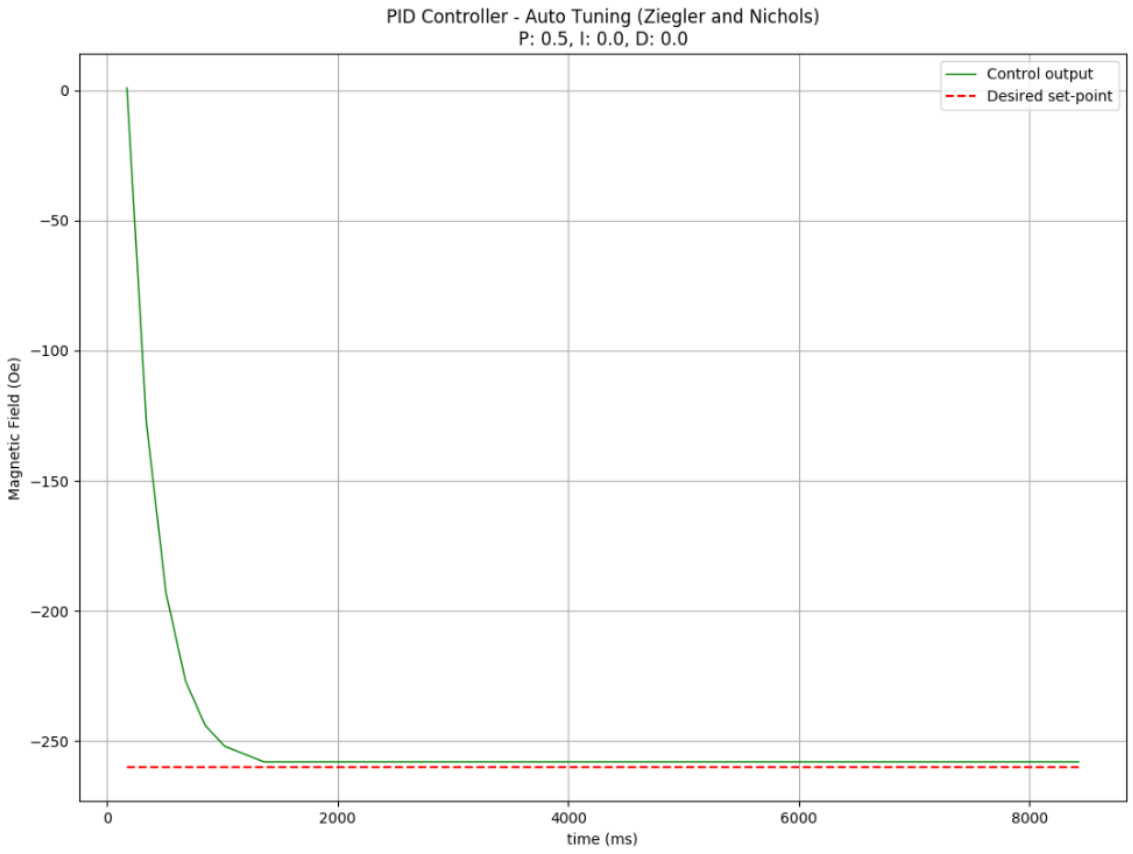


Figure 5.27: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. P-only type of controller with pre-set gain Kp = 0.5. Desired magnetic field is -260 Oe and starting magnetic field is 0 Oe. Initial error is -260 Oe, reached magnetic field is -258 Oe and settling time is 1700 ms. This tuning configuration outcomes steady-state error (or "droop").

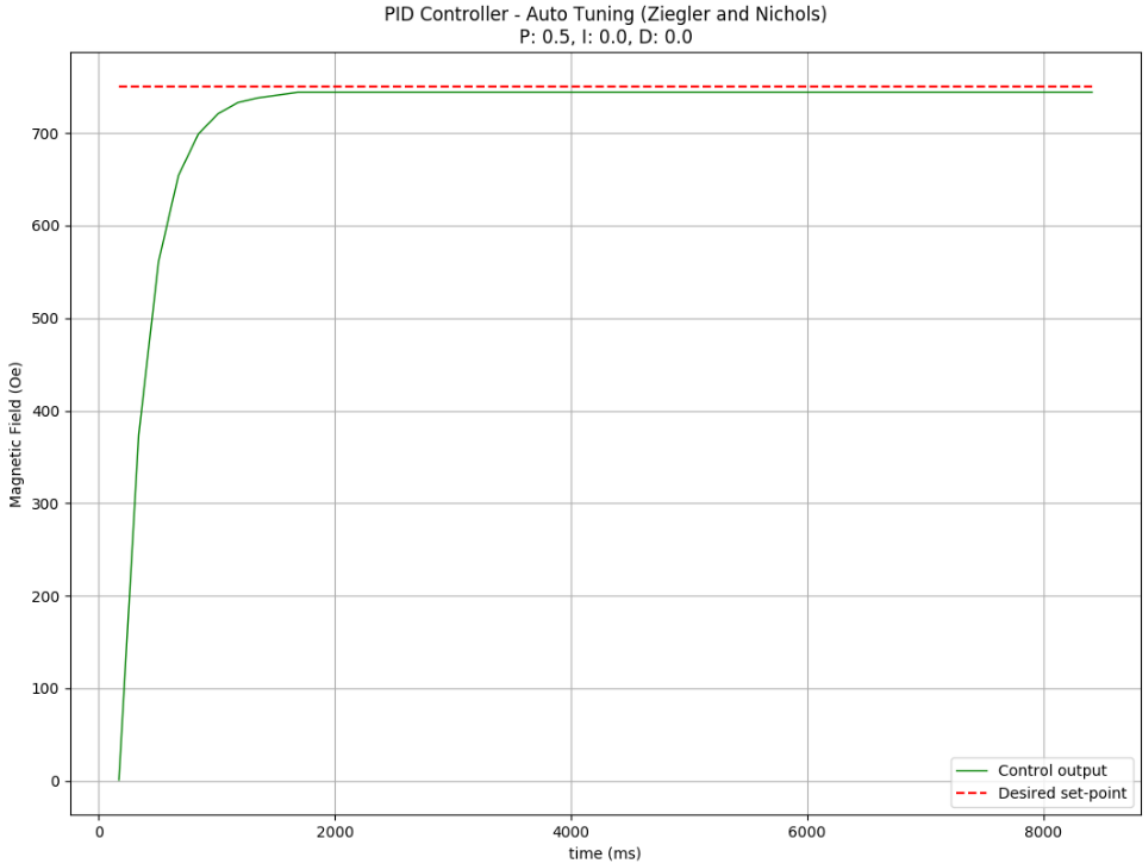


Figure 5.28: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. P-only type of controller with pre-set gain Kp = 0.5. Desired magnetic field is 750 Oe and starting magnetic field is 0 Oe. Initial error is 750 Oe, reached magnetic field is 744 Oe and settling time is 1800 ms. This tuning configuration outcomes steady-state error (or ”droop”).

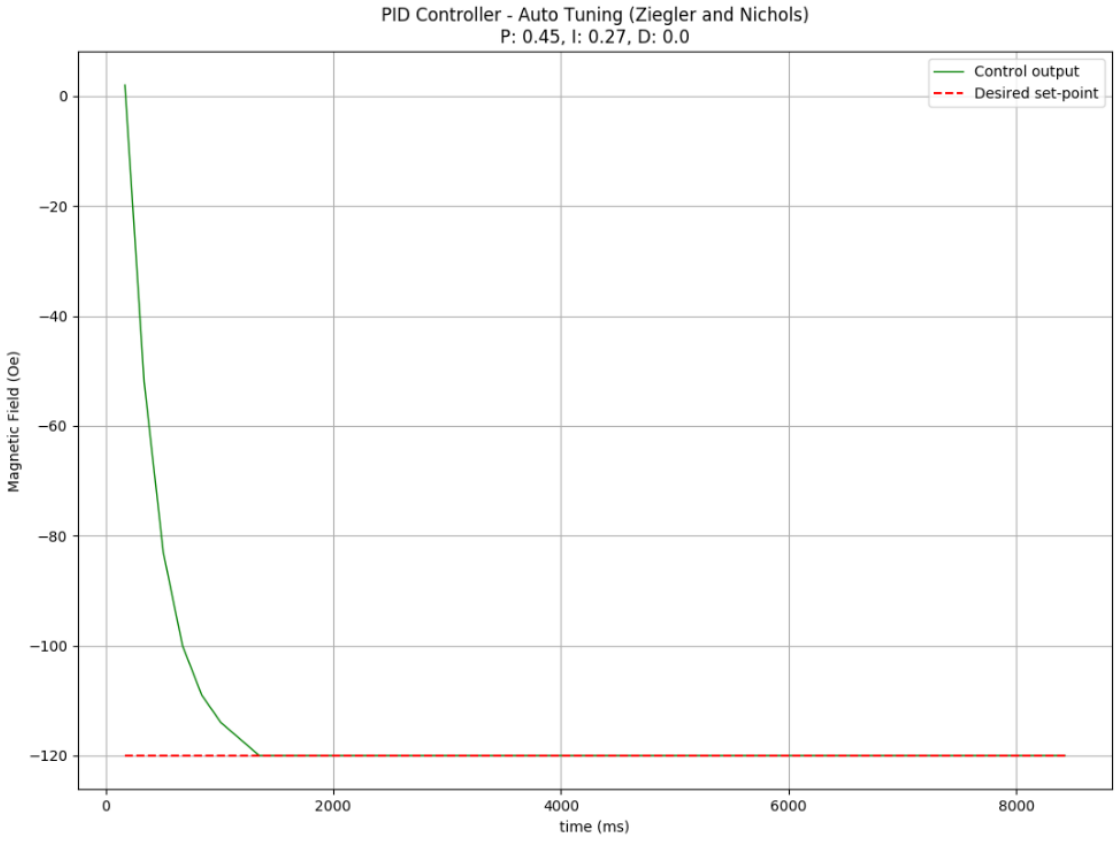


Figure 5.29: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is -120 Oe and starting magnetic field is 0 Oe. Initial error is -120 Oe, reached magnetic field is -120 Oe and settling time is 1200 ms. This tuning configuration fit-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting.

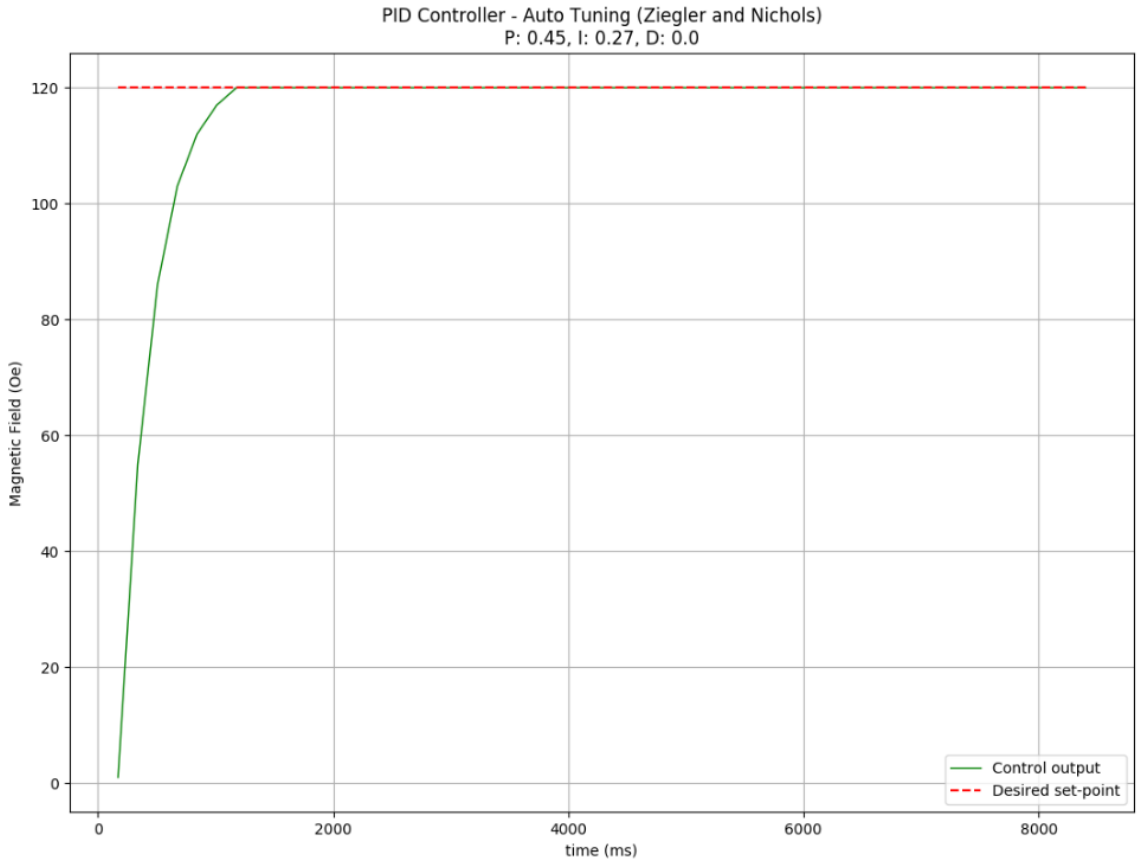


Figure 5.30: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is 120 Oe and starting magnetic field is 0 Oe. Initial error is 120 Oe, reached magnetic field is 120 Oe and settling time is 1000 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting.

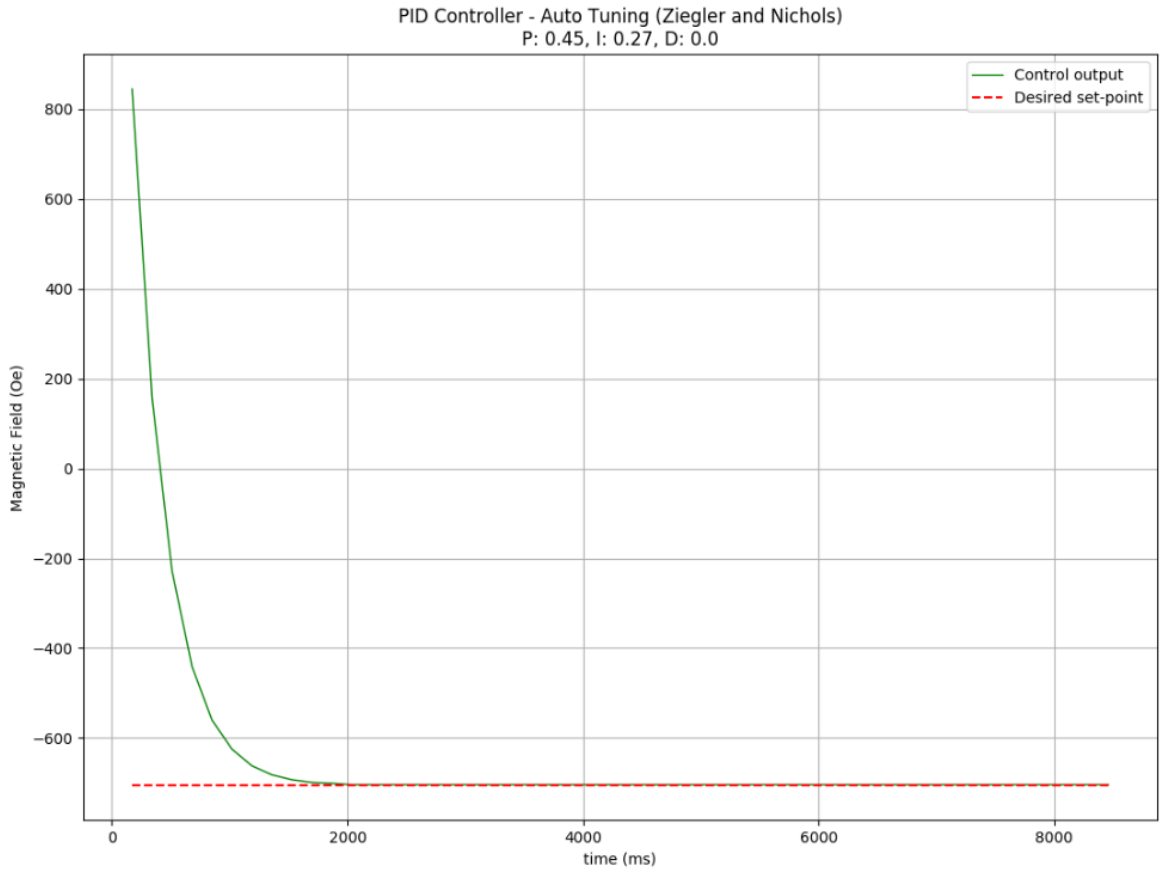


Figure 5.31: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is -705 Oe and starting magnetic field is 845 Oe. Initial error is -1550 Oe, reached magnetic field is -705 Oe and settling time is 2000 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting even when a very large change on the magnetic field (from 845 Oe to -705 Oe) is demanded by the end-user.

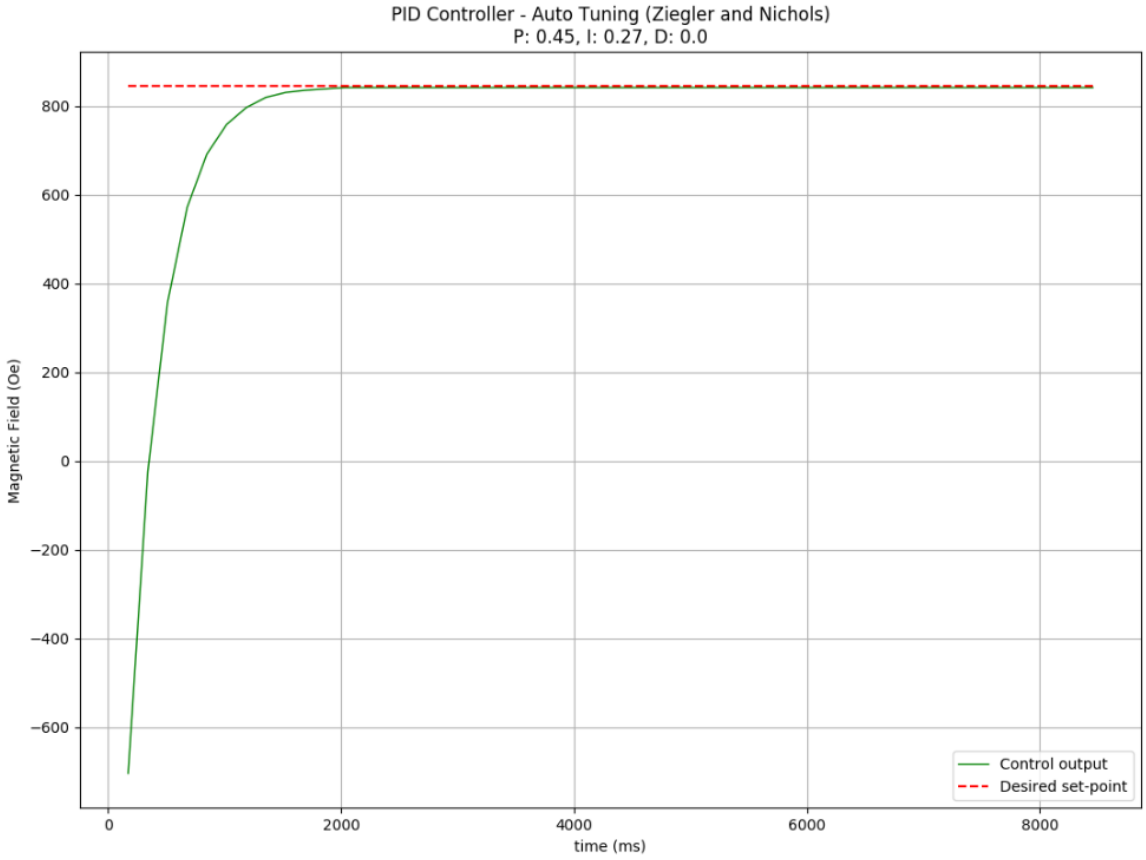


Figure 5.32: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is 845 Oe and starting magnetic field is -705 Oe. Initial error is 1550 Oe, reached magnetic field is 845 Oe and settling time is 2000 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting even when a very large change on the magnetic field (from -705 Oe to 845 Oe) is demanded by the end-user.

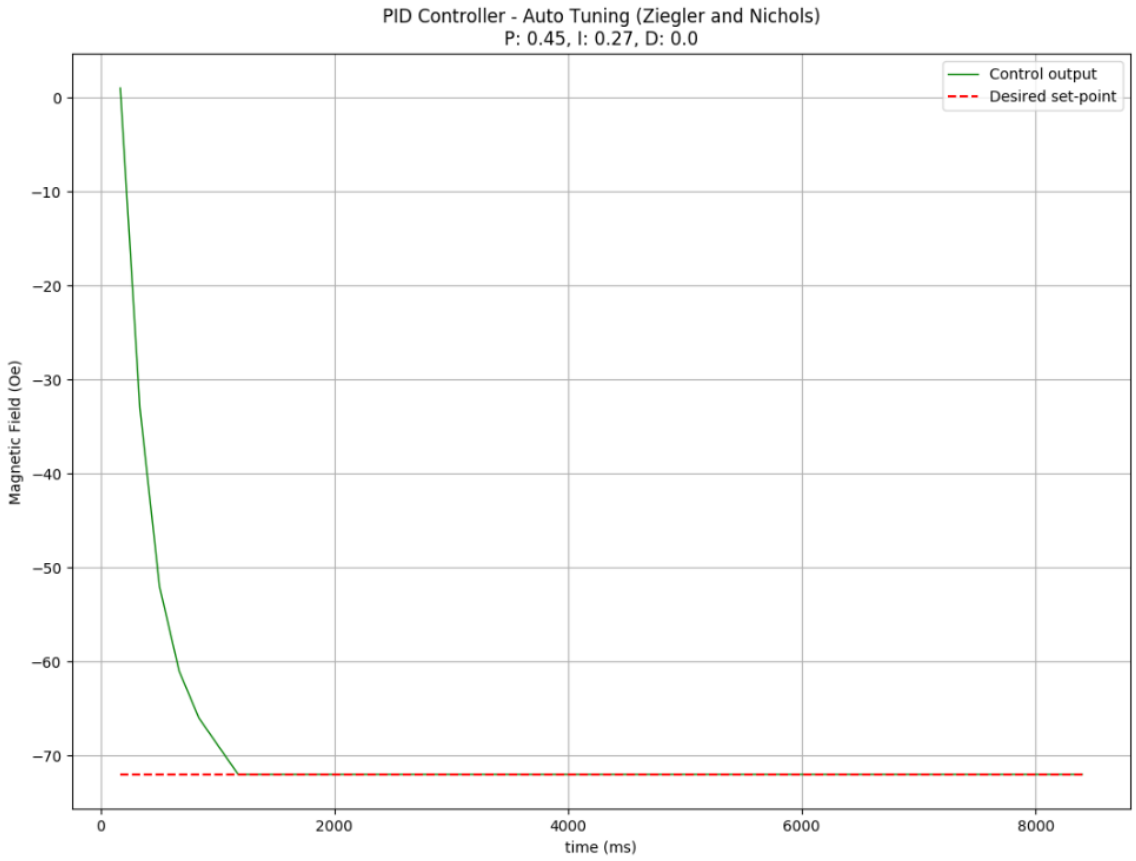


Figure 5.33: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is -72 Oe and starting magnetic field is 0 Oe. Initial error is -72 Oe, reached magnetic field is -72 Oe and settling time is 1000 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting even when a very small change on the magnetic field (from 0 Oe to -72 Oe) is demanded by the end-user.

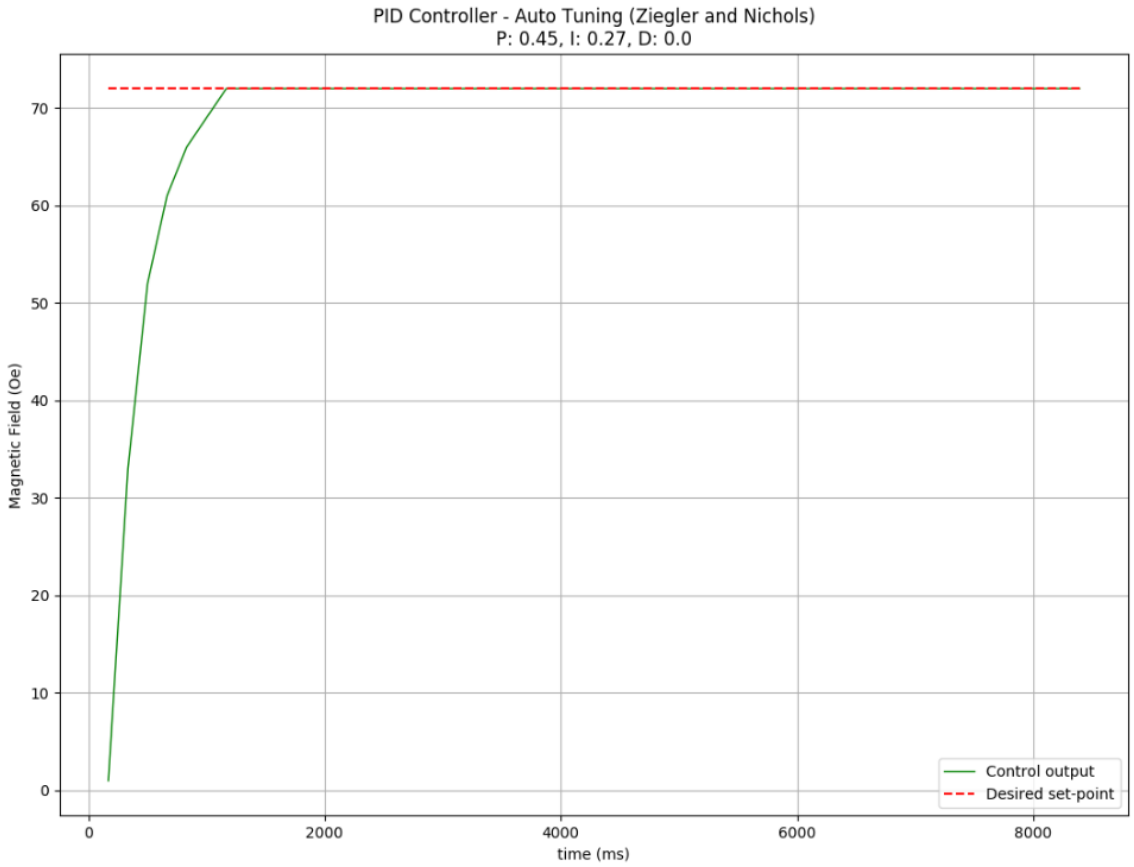


Figure 5.34: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.45 and Ki = 0.27. Desired magnetic field is 72 Oe and starting magnetic field is 0 Oe. Initial error is 72 Oe, reached magnetic field is 72 Oe and settling time is 1000 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting even when a very small change on the magnetic field (from 0 Oe to 72 Oe) is demanded by the end-user.

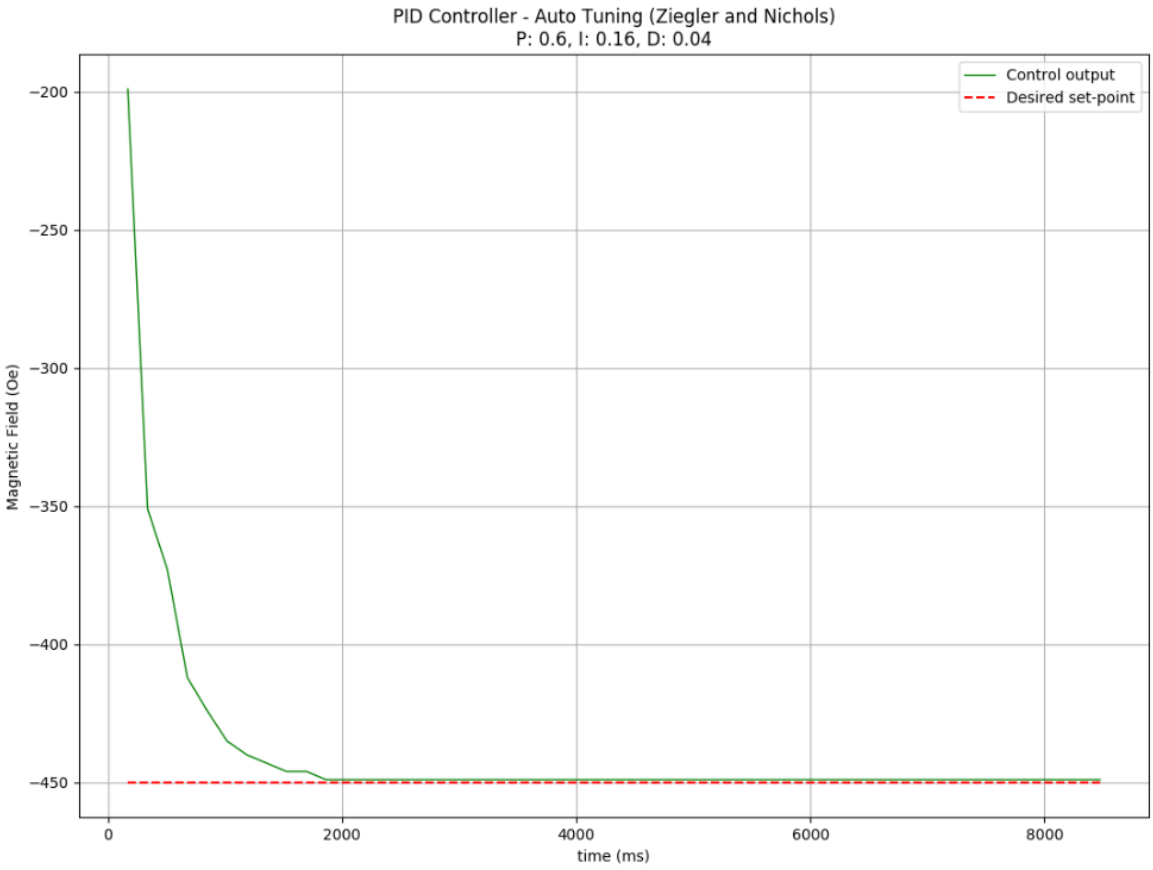


Figure 5.35: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.60, Ki = 0.16 and Kd = 0.04. Desired magnetic field is -450 Oe and starting magnetic field is -200 Oe. Initial error is -250 Oe, reached magnetic field is -448 Oe and settling time is 2000 ms. This tuning configuration outcomes (a small) steady-state error (or ”droop”). Signal is noisy due to derivative (D) term.

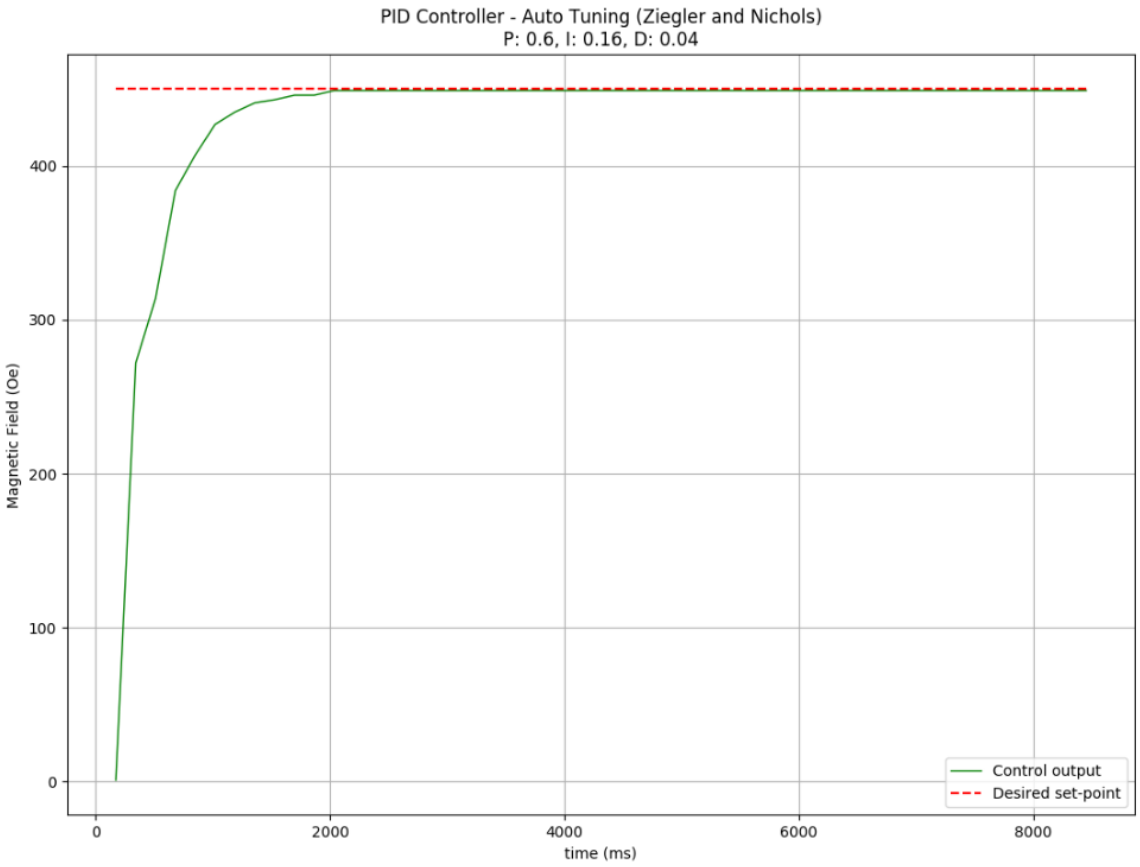


Figure 5.36: Auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.60, Ki = 0.16 and Kd = 0.04. Desired magnetic field is 450 Oe and starting magnetic field is 0 Oe. Initial error is 450 Oe, reached magnetic field is 448 Oe and settling time is 2000 ms. This tuning configuration outcomes (a small) steady-state error (or "droop"). Signal is noisy due to derivative (D) term.

Tyreus-Luyben Method

In order to conduct Tyreus-Luyben auto tuning method to the intelligent controller based on a Raspberry Pi, the same steps with Ziegler-Nichols method above have been followed, resulting with identical ultimate gain (Kpu = 1) and ultimate oscillation period (Tu = 0.32 second) values. However, in order to calculate the proportional (Kp), integral (Ki) and derivative (Kd) gains, the recorded ultimate gain (Kpu) and ultimate oscillation period (Tu) values had to be applied to the Tyreus-Luyben's recommended criteria shown in Table 2.3. Suggested PI, and PID control modes along with their Kp, Ki and Kd gains are PI (Kp = 0.31 and Ki = 0.70) and PID (Kp = 0.45, Ki = 0.70 and

Kd = 0.05). It is important that Tyreus-Luyben auto tuning method does not include the P-only control mode.

The following figures illustrate the Tyreus-Luyben' auto tuning tests conducted for the purposes of the intelligent controller based on a Raspberry Pi using PI (Kp = 0.31 and Ki = 0.70) and PID (Kp = 0.45, Ki = 0.70 and Kd = 0.05) tuning configurations. P, I and D values of the following graphs referred to as the proportional (Kp), integral (Ki) and derivative (Kd) gains. Full Tyreus-Luyben auto tuning tests log for the intelligent controller based on a Raspberry Pi system can be found in Appendix H-Table H.2.

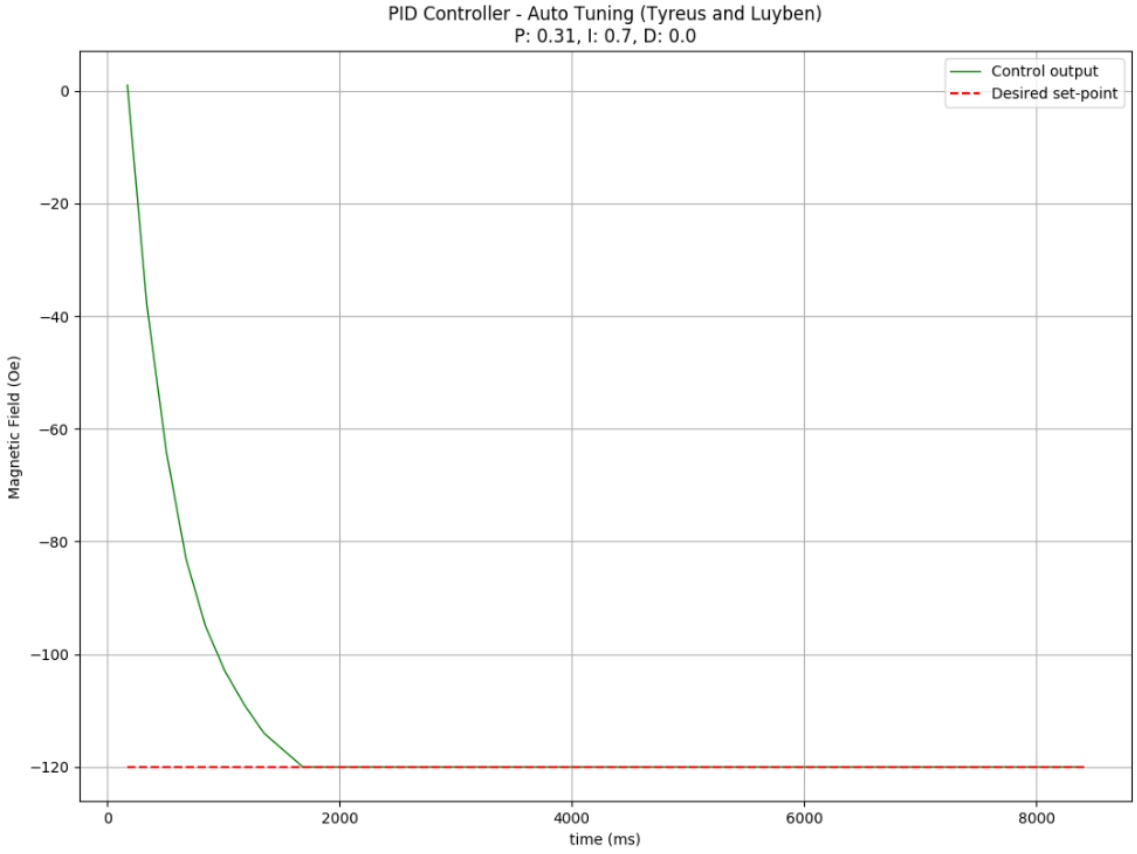


Figure 5.37: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is -120 Oe and starting magnetic field is 0 Oe. Initial error is -120 Oe, reached magnetic field is -120 Oe and settling time is 1800 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

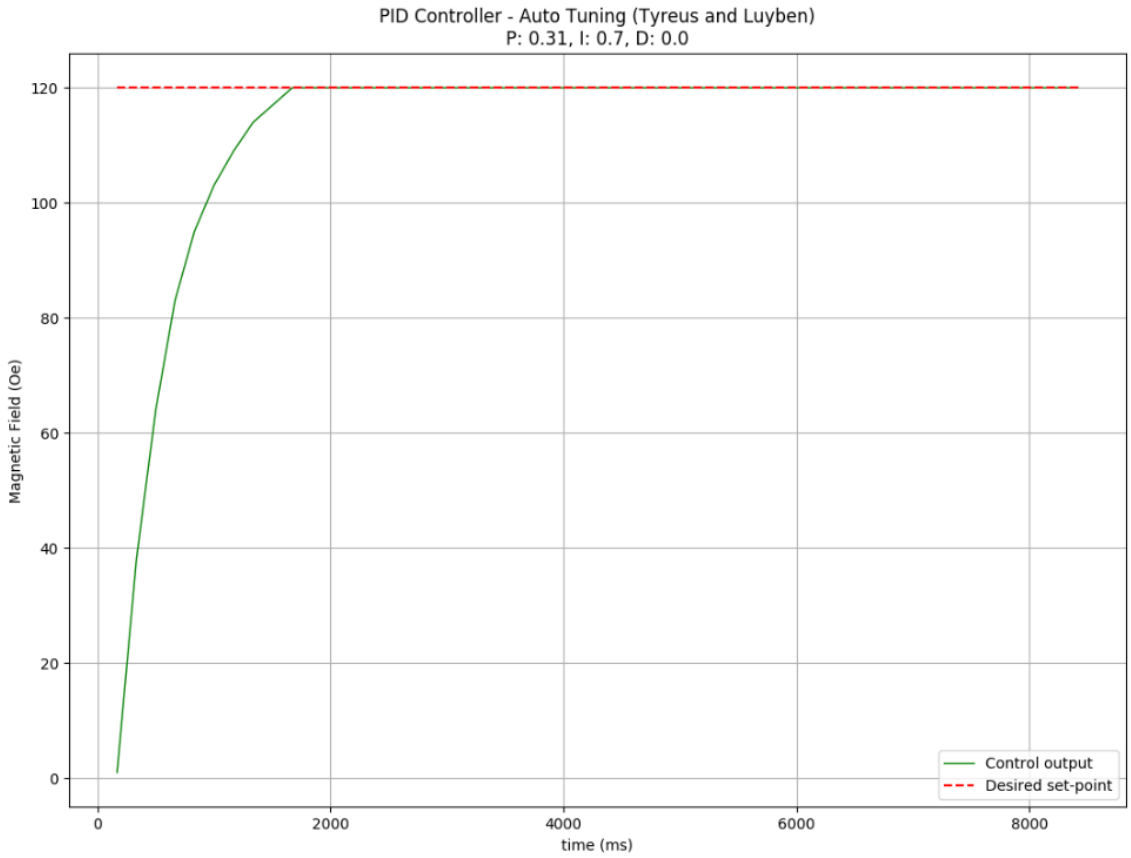


Figure 5.38: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is 120 Oe and starting magnetic field is 0 Oe. Initial error is 120 Oe, reached magnetic field is 120 Oe and settling time is 1800 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

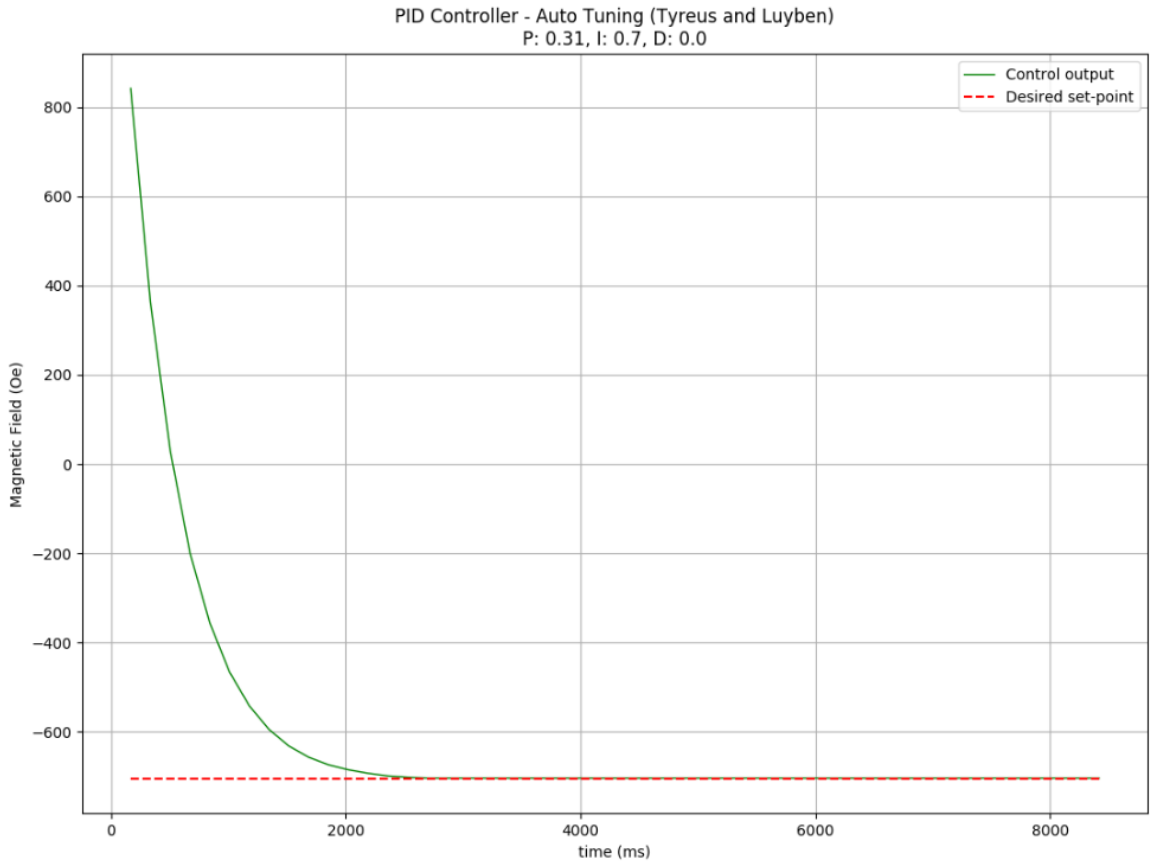


Figure 5.39: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is -705 Oe and starting magnetic field is 842 Oe. Initial error is -1547 Oe, reached magnetic field is -705 Oe and settling time is 2800 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting even when a very large change on the magnetic field (from -705 Oe to 842 Oe) is demanded by the end-user. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

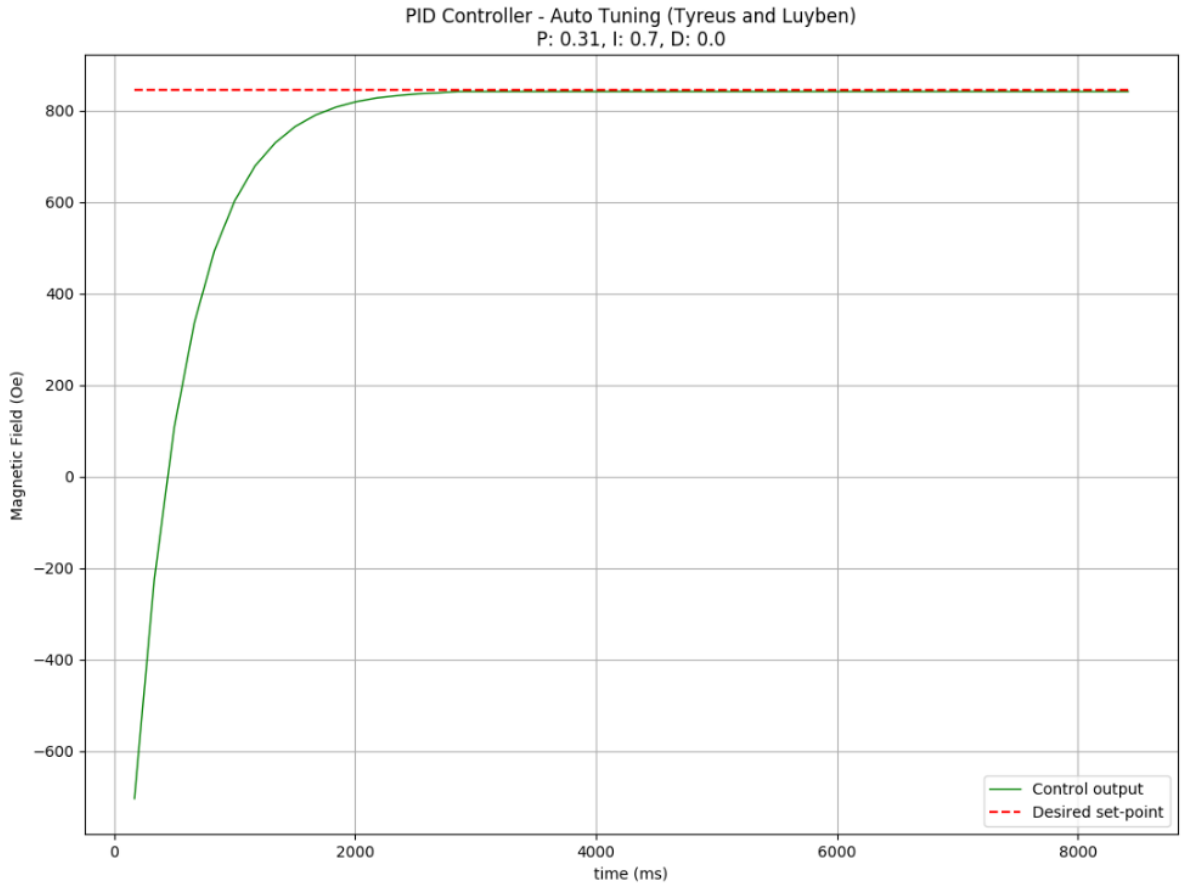


Figure 5.40: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is 845 Oe and starting magnetic field is -704 Oe. Initial error is -1549 Oe, reached magnetic field is 845 Oe and settling time is 3100 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting even when a very large change on the magnetic field (from 842 Oe to -705 Oe) is demanded by the end-user. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

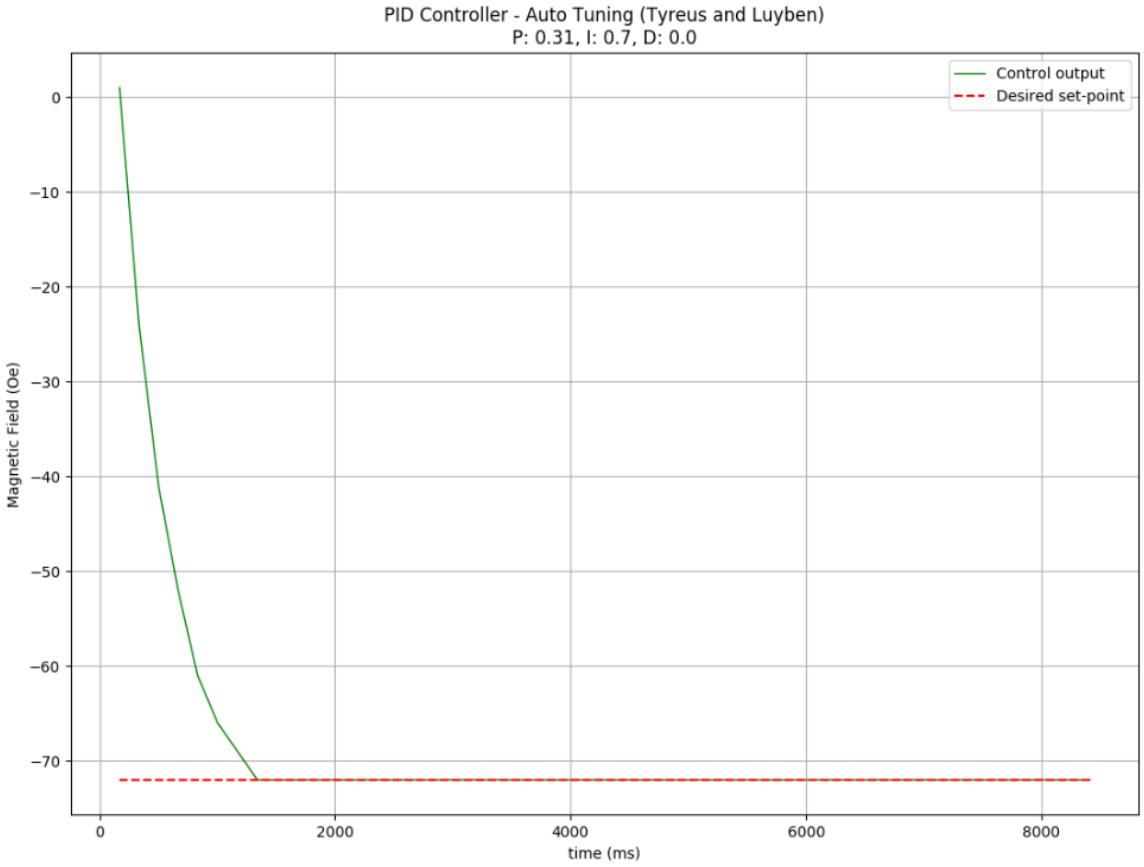


Figure 5.41: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is -72 Oe and starting magnetic field is 0 Oe. Initial error is -72 Oe, reached magnetic field is -72 Oe and settling time is 1500 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of undershooting even when a very small change on the magnetic field (from 0 Oe to -72 Oe) is demanded by the end-user. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

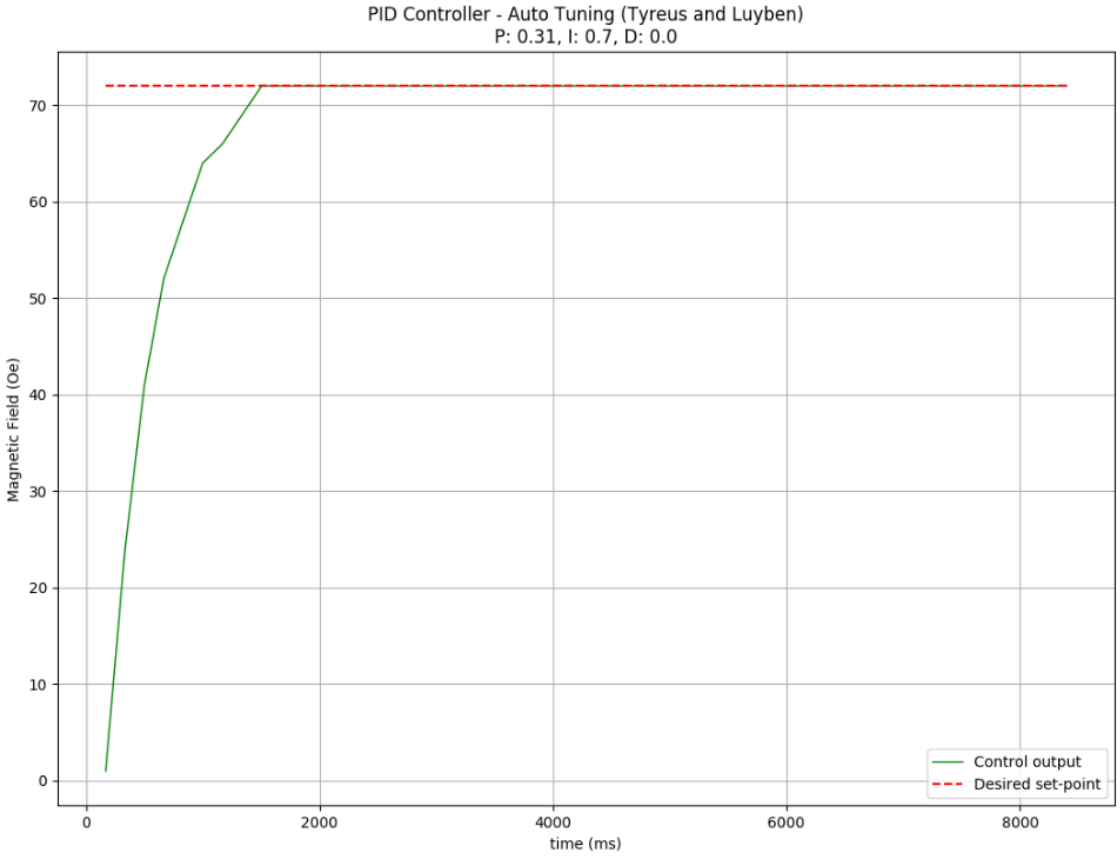


Figure 5.42: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.31 and Ki = 0.70. Desired magnetic field is 72 Oe and starting magnetic field is 0 Oe. Initial error is 72 Oe, reached magnetic field is 72 Oe and settling time is 1500 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi. Outcome of the controlled process is stable, accurate and fast, without any signs of overshooting even when a very small change on the magnetic field (from 0 Oe to 72 Oe) is demanded by the end-user. However, this tuning configuration is slightly slower than the Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) optimal tuning configuration.

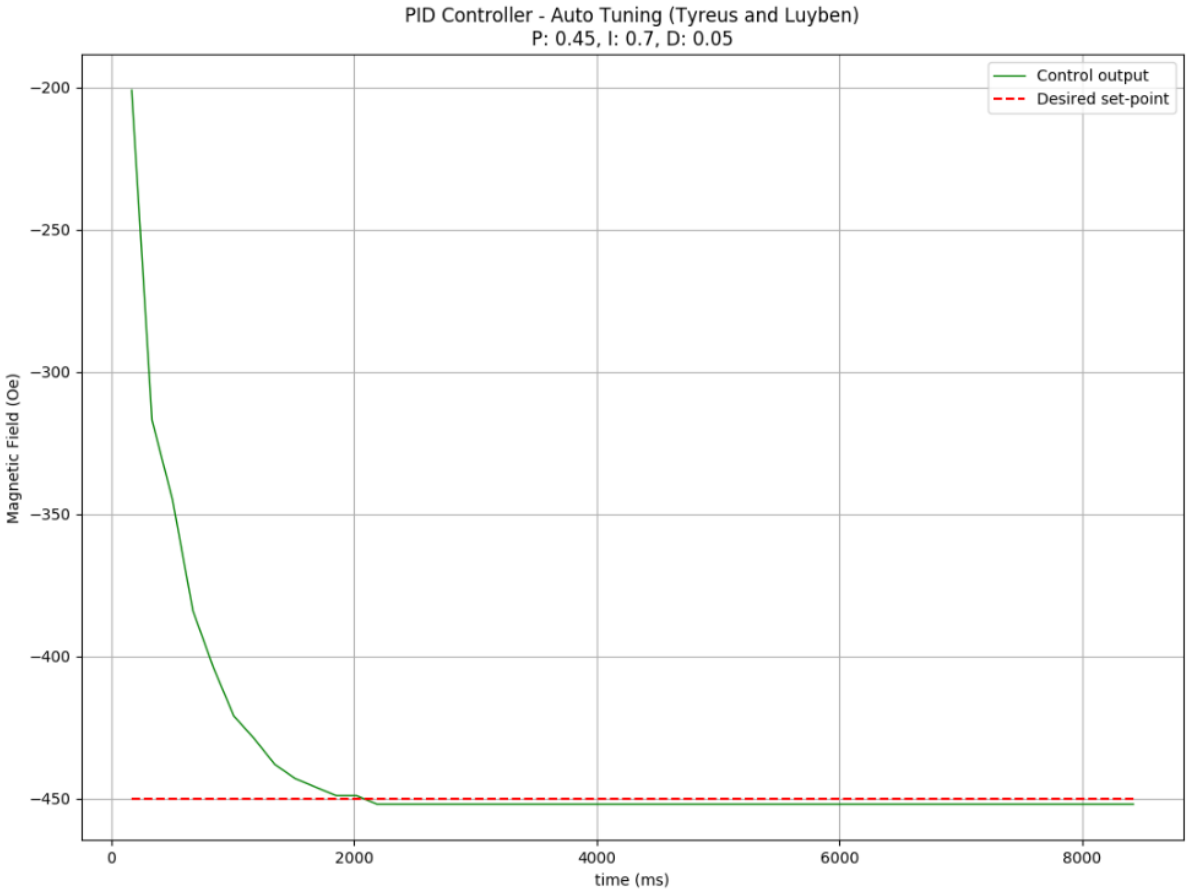


Figure 5.43: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.45, Ki = 0.70 and Kd = 0.05. Desired magnetic field is -450 Oe and starting magnetic field is -202 Oe. Initial error is -248 Oe, reached magnetic field is -458 Oe and settling time is 2200 ms. This tuning configuration outcomes (a small) signs of undershooting. Signal is noisy due to derivative (D) term.

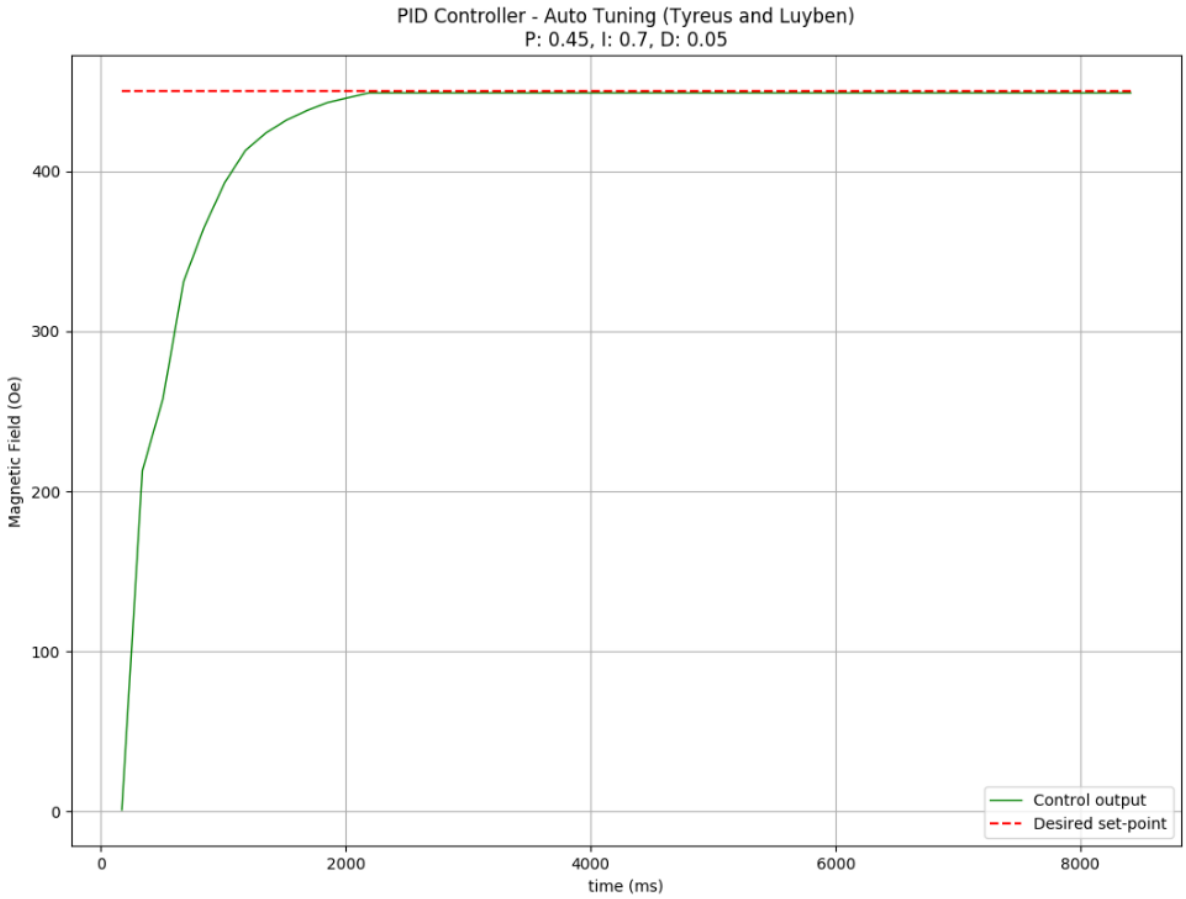


Figure 5.44: Auto tuning (Tyreus-Luyben method) of the intelligent controller based on a Raspberry Pi. PID type of controller with pre-set gains Kp = 0.45, Ki = 0.70 and Kd = 0.05. Desired magnetic field is 450 Oe and starting magnetic field is 0 Oe. Initial error is 450 Oe, reached magnetic field is 448 Oe and settling time is 2200 ms. This tuning configuration outcomes (a small) steady-state error (or ”droop”). Signal is noisy due to derivative (D) term.

5.4 Results Analysis, Discussion and Evaluation

First, hardware unit testing verified the initial communication establishment with both gaussmeter and power supply, using Lake Shore communication and NI VISA interactive control tools, accordingly. Afterwards, software unit testing approved that all the individual features offered by the custom-initiated UI are user-friendly and functioning properly. Features of the terminal-based UI include *Exit Program*, *Run Program*, *Calibrate for different magnet* and *Enable/Disable plotting*.

In addition, for the integrated system testing purposes, different manual tuning configurations have been investigated by setting Kp, Ki and Kd gains for the proportional (P), integral (I) and derivative (D) modes, respectively. Manual tuning of the intelligent controller based on a Raspberry Pi was useful for some initial observations of how the PID control process is working, and thus, measure different performance indices. For example, ”anti-windup” feature of the software has been intentionally excluded from a number of manual tuning tests in order to investigate any possible signs of either overshooting or undershooting in the system. Figures 5.16 and 5.17 show undershooting and overshooting in the system, due to ”integrator windup” problem. Figures 5.18 and 5.19 illustrate how these undershooting and overshooting issues have been resolved, using ”anti-windup” software feature in the form of pre-defined set-point guards. However, as it is based on trial and error procedure, manual tuning is time-consuming and requires experience. Therefore, auto tuning methods of Ziegler-Nichols and Tyreus-Luyben have been used to find the fit-for-purpose tuning configuration(s) for the intelligent controller based on a Raspberry Pi.

After analyzing and evaluating the test results of different auto tuning configurations (Nichols-Ziegler and Tyreus-Luyben methods), the fit-for-purpose configurations have been shortlisted. Table 5.1 summarizes the integrated system testing results for the shortlisted Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) and Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configurations which are applicable for the purposes of the intelligent controller based on a Raspberry Pi. It is important that both P-only and PID type of controllers have been discarded for the purposes of the intelligent controller based on a Raspberry Pi. This is due to steady-state error (or ”droop”) and high-frequency noise outcomes in both Ziegler-Nichols and Tyreus-Luyben auto tuning configurations (see Figures 5.27, 5.28, 5.35, 5.36, 5.43 and 5.44). Steady-state error experienced using a P-only type of controller, while high-frequency noise observed due to the inclusion of the derivative (D) term, forming a PID type of controller.

Tuning method	Type of controller	Kp	Ki	Desired MF (Oe)	Starting MF (Oe)	Reached MF (Oe)	Settling time (ms)	Figure
ZN	PI	0.45	0.27	-120	0	-120	1200	5.29
ZN	PI	0.45	0.27	120	0	120	1000	5.30
ZN	PI	0.45	0.27	-705	845	-705	2000	5.31
ZN	PI	0.45	0.27	845	-705	845	2000	5.32
ZN	PI	0.45	0.27	-72	0	-72	1000	5.33
ZN	PI	0.45	0.27	72	0	72	1000	5.34
TL	PI	0.31	0.70	-120	0	-120	1800	5.37
TL	PI	0.31	0.70	120	0	120	1800	5.38
TL	PI	0.31	0.70	-705	842	-705	2800	5.39
TL	PI	0.31	0.70	845	-704	845	3100	5.40
TL	PI	0.31	0.70	-72	0	-72	1500	5.41
TL	PI	0.31	0.70	72	0	72	1500	5.42

Table 5.1: Integrated system testing results for the shortlisted Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) and Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configurations which are applicable for the purposes of the intelligent controller based on a Raspberry Pi. ZN refers to as the Ziegler-Nichols, TL refers to as the Tyreus-Luyben, PI refers to as the Proportional-Integral, Kp refers to as the proportional gain, Ki refers to as the integral gain and MF refers to as the Magnetic Field.

Generally, the integrated system testing results of the shortlisted Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) and Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configurations denoted that the prompted desire magnetic fields have been successfully reached using an intelligent PI (Proportional-Integral) controller based on a Raspberry Pi. For a thorough performance comparison between the shortlisted Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) and Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configurations, it was important to prompt the same (or close enough) desired magnetic fields for any dual testing combinations. Similarly, starting magnetic fields, and thus, initial error, also kept the same (or close enough) for any dual testing combinations of the shortlisted Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) and Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configurations.

In terms of the system performance, results of the PI type Ziegler-Nichols and Tyreus-Luyben auto tuning configurations showed that the controlled process is stable,

accurate and fast, without any signs of either overshooting or undershooting. Repeatability of the intelligent controller based on a Raspberry Pi for PI type Ziegler-Nichols and Tyreus-Luyben auto tuning configurations can also be verified as the system has been tested for different desired magnetic field inputs, including very large and small changes in relation with the starting magnetic fields. However, as it can be remarked from the Table 5.1, Ziegler-Nichols (PI - Kp = 0.45 and Ki = 0.27) auto tuning configuration is faster (less settling time) for all testing combinations, comparing to the Tyreus-Luyben (PI - Kp = 0.31 and Ki = 0.70) auto tuning configuration. Therefore, PI type Ziegler-Nichols auto tuning configuration with gains of Kp = 0.45 and Ki = 0.27 has been used to conduct further speed optimization in the system. Speed optimization for the purposes of the intelligent PI (Proportional-Integral) controller based on a Raspberry Pi is discussed in section 5.5.

5.5 Speed Optimization

The integrated system testing results concluded that the Ziegler-Nichols auto tuning configuration with gains of Kp = 0.45 and Ki = 0.27 is the most applicable for the intelligent controller based on a Raspberry Pi. Therefore, further speed optimization has been conducted in the system, for a PI type intelligent controller, using Ziegler-Nichols auto tuning method criteria (see Table 2.2). It is important that with the current optimum auto tuning configuration (Kp = 0.45 and Ki = 0.27), the system is stable, accurate and fast enough, without any signs of either overshooting or undershooting. Thus, ultimate goal of the speed optimization is not only to enhance the system's response time, but also to keep stability and accuracy measures at optimum levels, without introducing any overshooting or undershooting. This is because stability, absence of either overshooting or undershooting and accuracy are considered as higher-priority performance indices than the speed (or settling time) of the proposed magnetic field closed-loop control system (see section 5).

Before performing the speed optimization for the purposes of the intelligent controller based on a Raspberry Pi, time limitations of the system had to be considered. In particular, time delays in the system include 0.1 second for each iteration of the PID control loop, 0.03 second for a serial communication with the Lake Shore 425 gaussmenter and 0.03 second for a GPIB communication with the Kepco BOP power supply. In particular, for the serial communication with the Lake Shore 425 gaussmeter, there is a reading rate limitation of 30 times per second (see section 3.2.1). Therefore, serial

communication time delay of 0.03 second kept the same. GPIB communication delay of 0.03 second also kept unchanged. On the other hand, there was a room for time improvement in the PID control loop, where time delay of 0.1 second for each iteration reduced down to 0.05 second in order to perform speed optimization in the system.

In order to conduct speed optimization for the purposes of the intelligent controller based on a Raspberry Pi, Ziegler-Nichols auto tuning method had to be applied again. This is because timings in the system are now different, and thus, ultimate gain (Kpu), as well as, ultimate oscillation period (Tu) are expected to be different. After performing the recommended steps of the Ziegler-Nichols auto tuning method (see section 5.3.2), Figure 5.45 illustrates the intelligent controller based on a Raspberry Pi to go into sustained periodic oscillation when the proportional (Kp) gain is set to 1.1. Therefore, ultimate gain (Kpu) is equals to 1.1 as well. In order to measure the ultimate oscillation period (Tu) some calculations required. In particular, as it can be remarked from Figure 5.45, the PID control process is running for $\sim$5.5 seconds by the time it first reaches the desired magnetic field set-point (200 Oe). PID control process is also pre-determined to run for 50 iterations (see section 5.3). Therefore, the following equations resulted with the ultimate oscillation period (Tu) for the intelligent controller based on a Raspberry Pi:

$$Ti = executionTime/numOfIterations \sim= 5.5/50 \sim= 0.11\ second, \tag{5.3}$$

where i is 1, T1 is the time period of a single oscillation (or cycle), *executionTime* is the overall time taken to run PID control process and *numOfIterations* is the pre-set number of iterations for the executed control loop. Therefore, as the ultimate oscillation period (Tu) is referred to as the time captured for two sustained oscillations (or cycles):

$$Tu = T1 * 2 \sim= 0.22\ second\ (or\ 220\ ms) \tag{5.4}$$

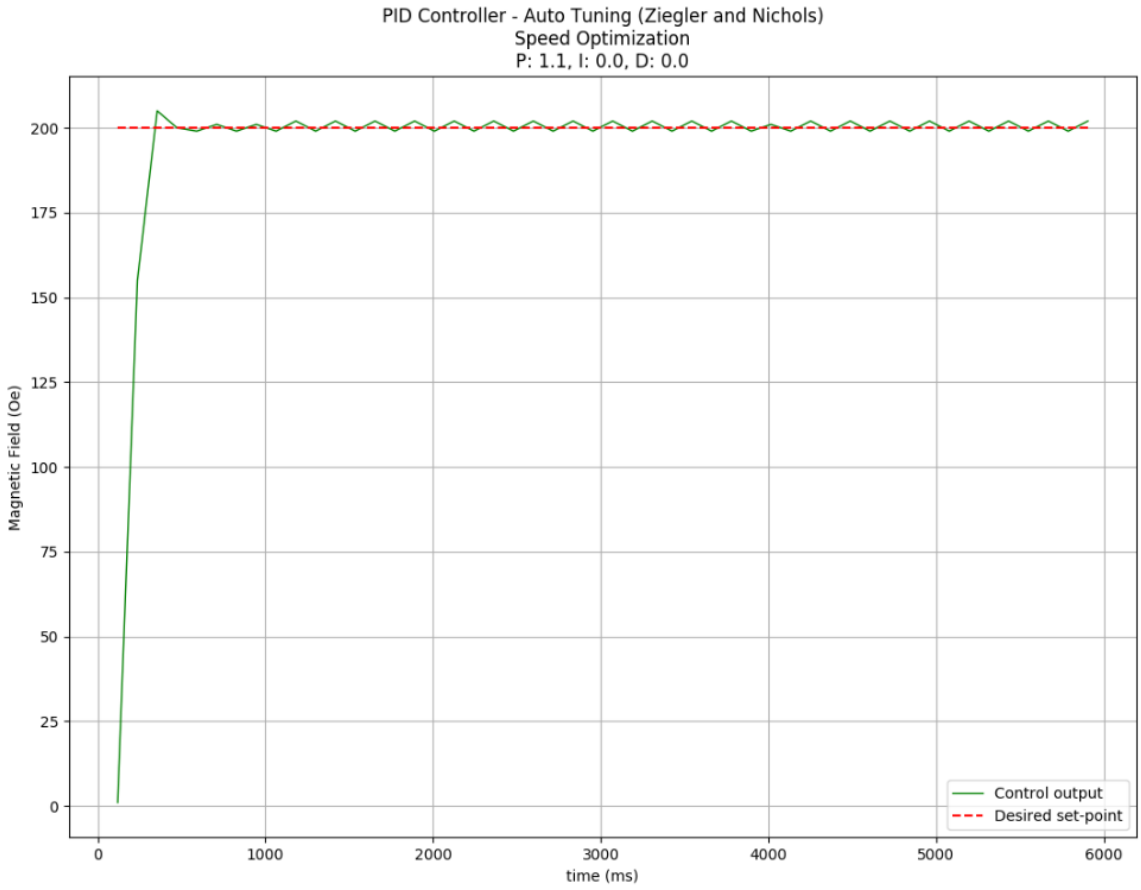


Figure 5.45: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. System goes into sustained periodic oscillation when the proportional (Kp) gain is set to 1.1.

The extracted ultimate gain (Kpu) and ultimate oscillation period (Tu) values were then used to calculate the proportional (Kp) and integral (Ki) gains, using Ziegler-Nichols recommended criteria of Table 2.2. Speed optimized PI control mode along with its Kp and Ki gains is PI (Kp = 0.5 and Ki = 0.18). It is important that P-only and PID control modes have not been considered for the speed optimization as it was previously evaluated from the integrated system tests that PI type of controller is the most suitable for this project (see section 5.4).

The following figures illustrate the speed optimized Ziegler-Nichols' auto tuning tests performed for the purposes of the intelligent controller based on a Raspberry Pi using PI (Kp = 0.5, Ki = 0.18 and Kd = 0) tuning configuration. P, I and D values of the following graphs referred to as the proportional (Kp), integral (Ki) and derivative (Kd) gains.

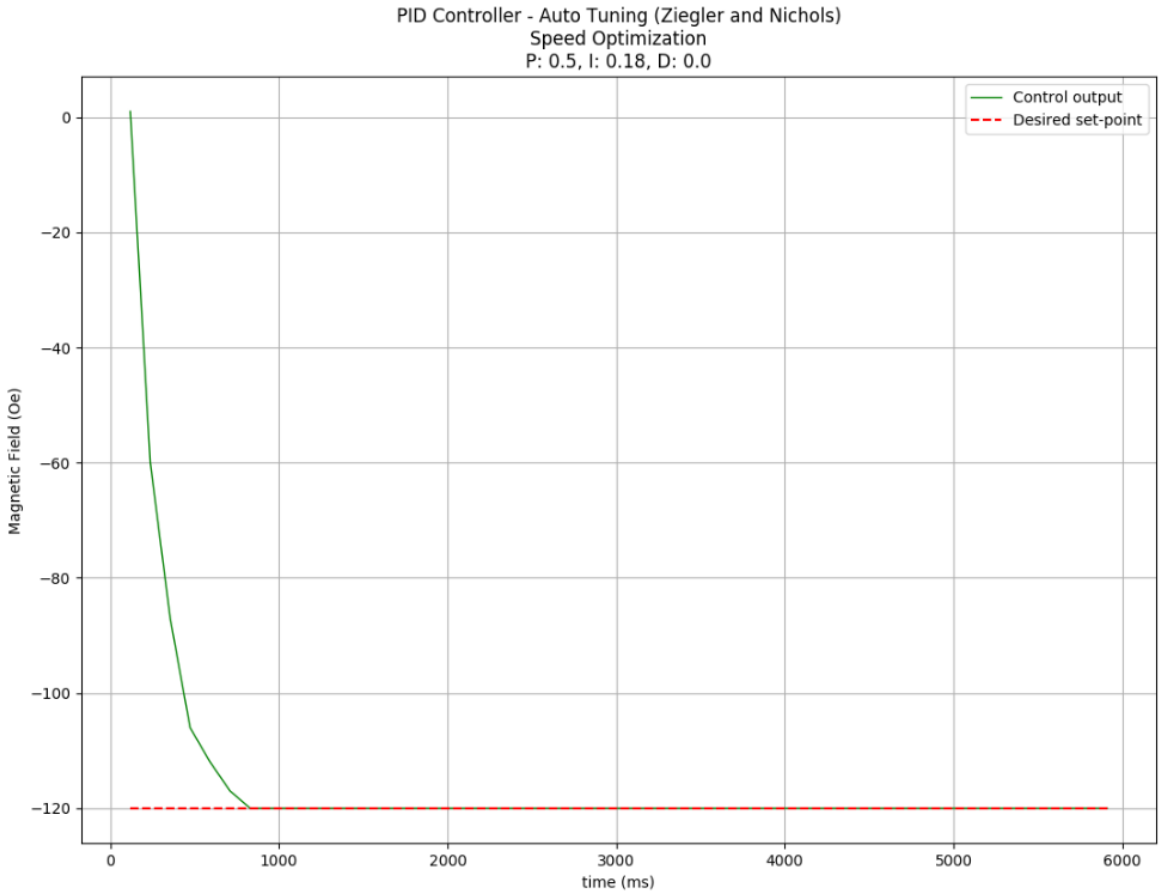


Figure 5.46: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is -120 Oe and starting magnetic field is 0 Oe. Initial error is -120 Oe, reached magnetic field is -120 Oe and settling time is 800 ms. This tuning configuration fit-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration.

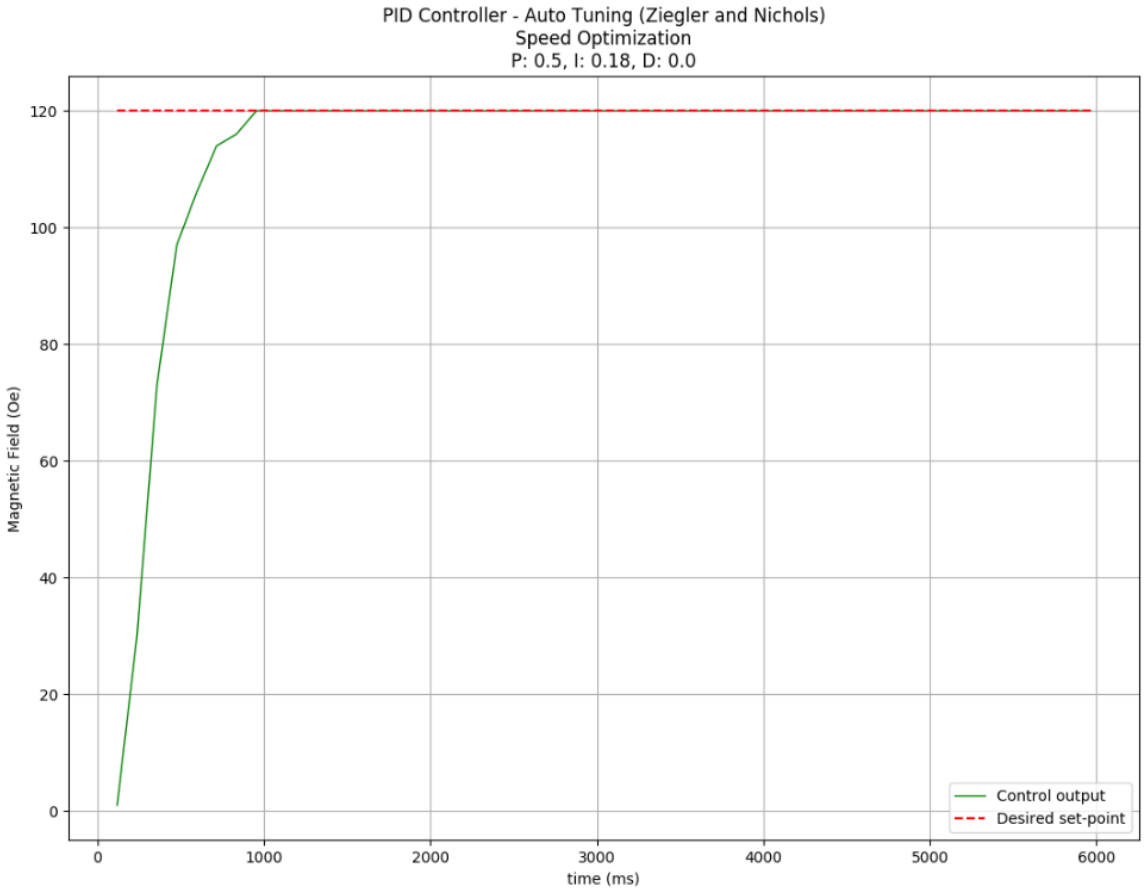


Figure 5.47: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is 120 Oe and starting magnetic field is 0 Oe. Initial error is 120 Oe, reached magnetic field is 120 Oe and settling time is 950 ms. This tuning configuration fit-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration.

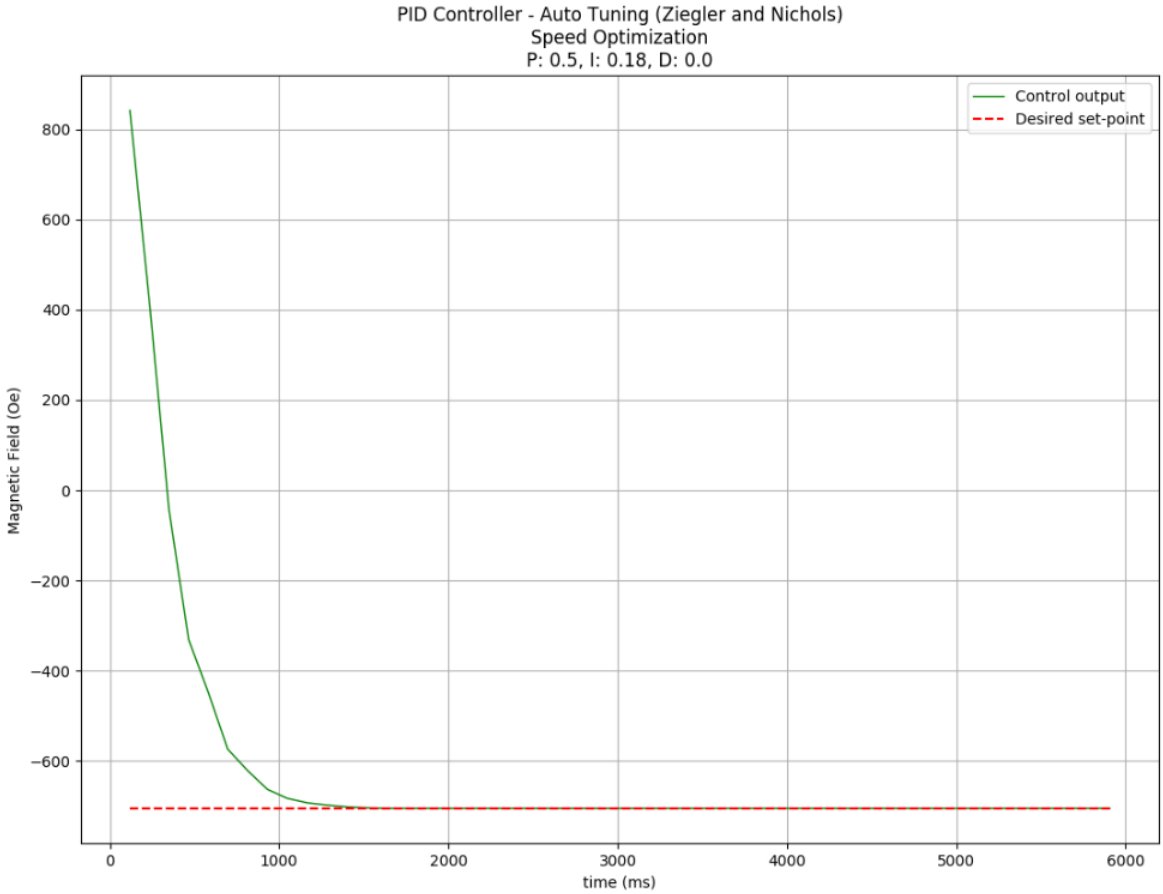


Figure 5.48: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is -705 Oe and starting magnetic field is 845 Oe. Initial error is -1550 Oe, reached magnetic field is -705 Oe and settling time is 1400 ms. This tuning configuration fit-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration.

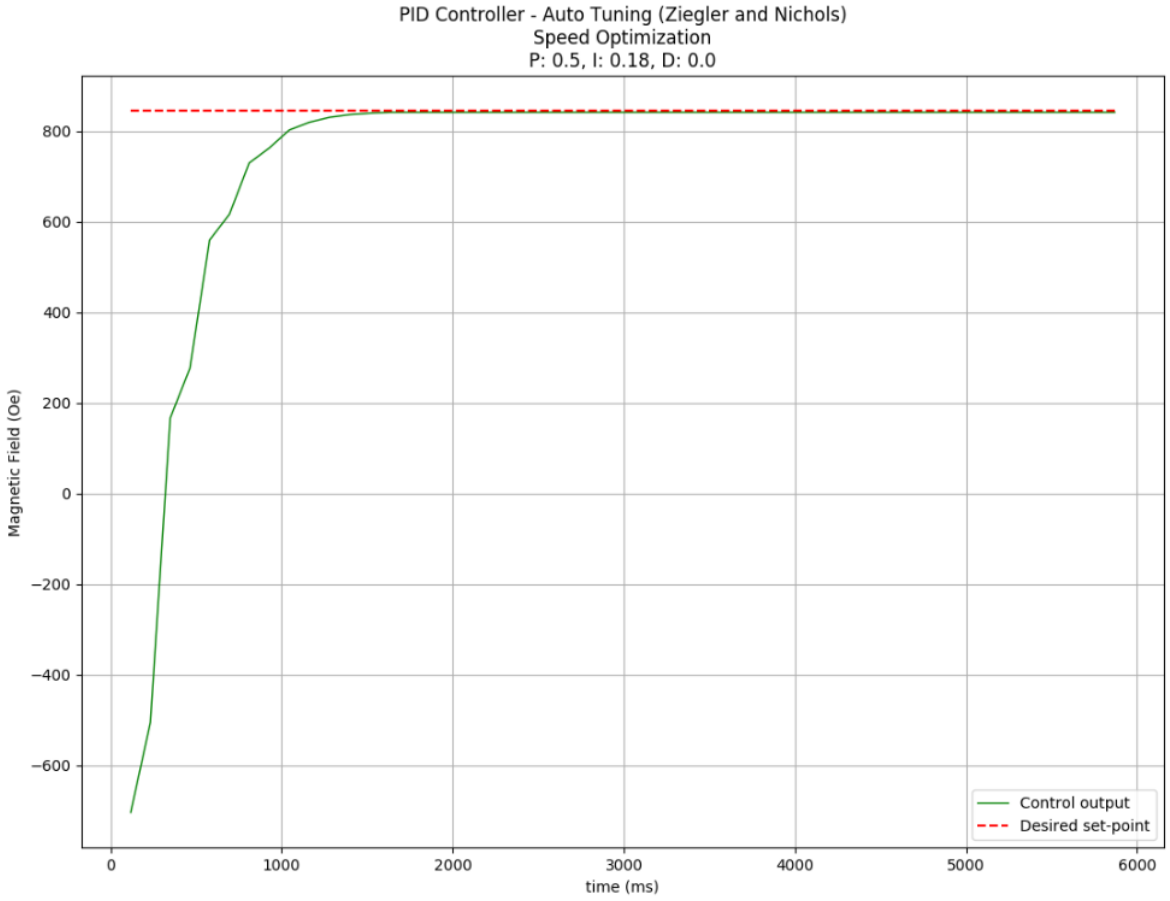


Figure 5.49: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is 845 Oe and starting magnetic field is -704 Oe. Initial error is 1549 Oe, reached magnetic field is 845 Oe and settling time is 1600 ms. This tuning configuration fit-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration. However, signal is noisy due to very large change on the magnetic field (from -704 Oe to 845 Oe).

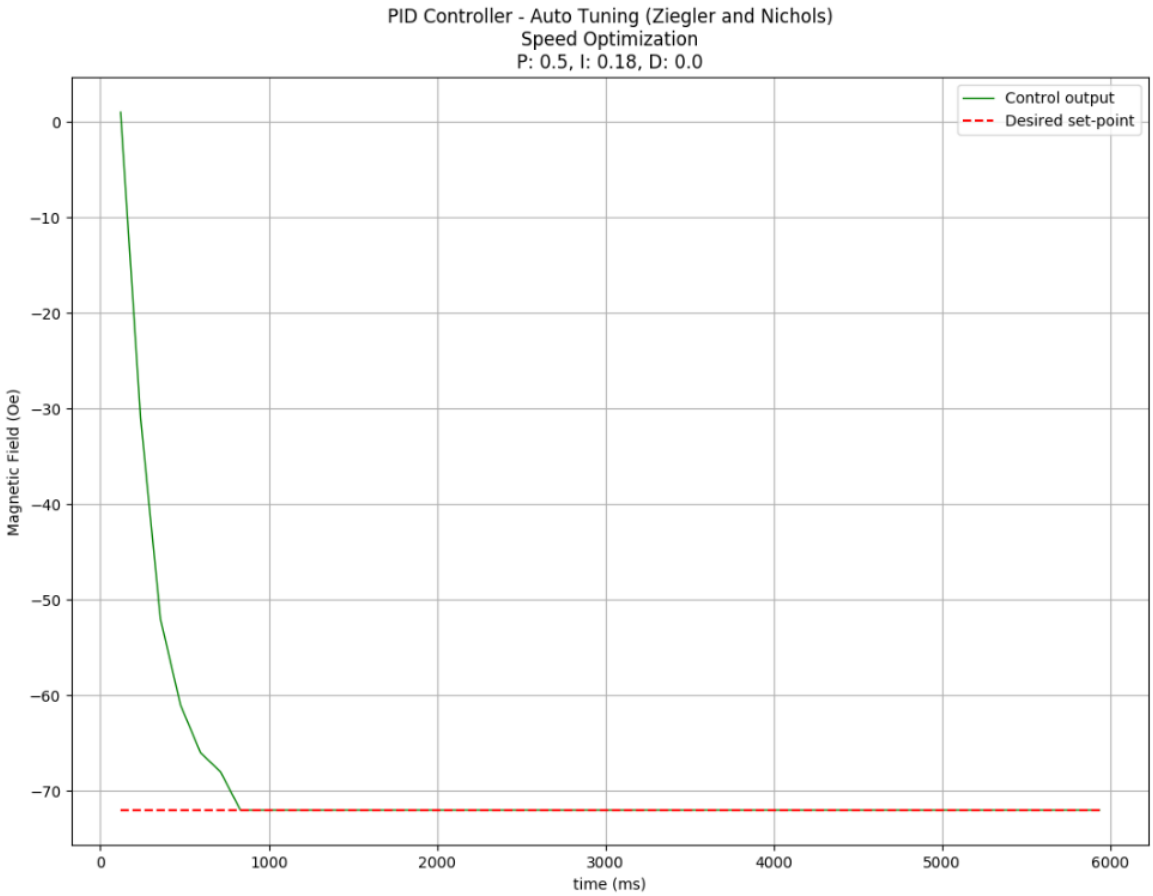


Figure 5.50: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is -72 Oe and starting magnetic field is 0 Oe. Initial error is -72 Oe, reached magnetic field is -72 Oe and settling time is 800 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration.

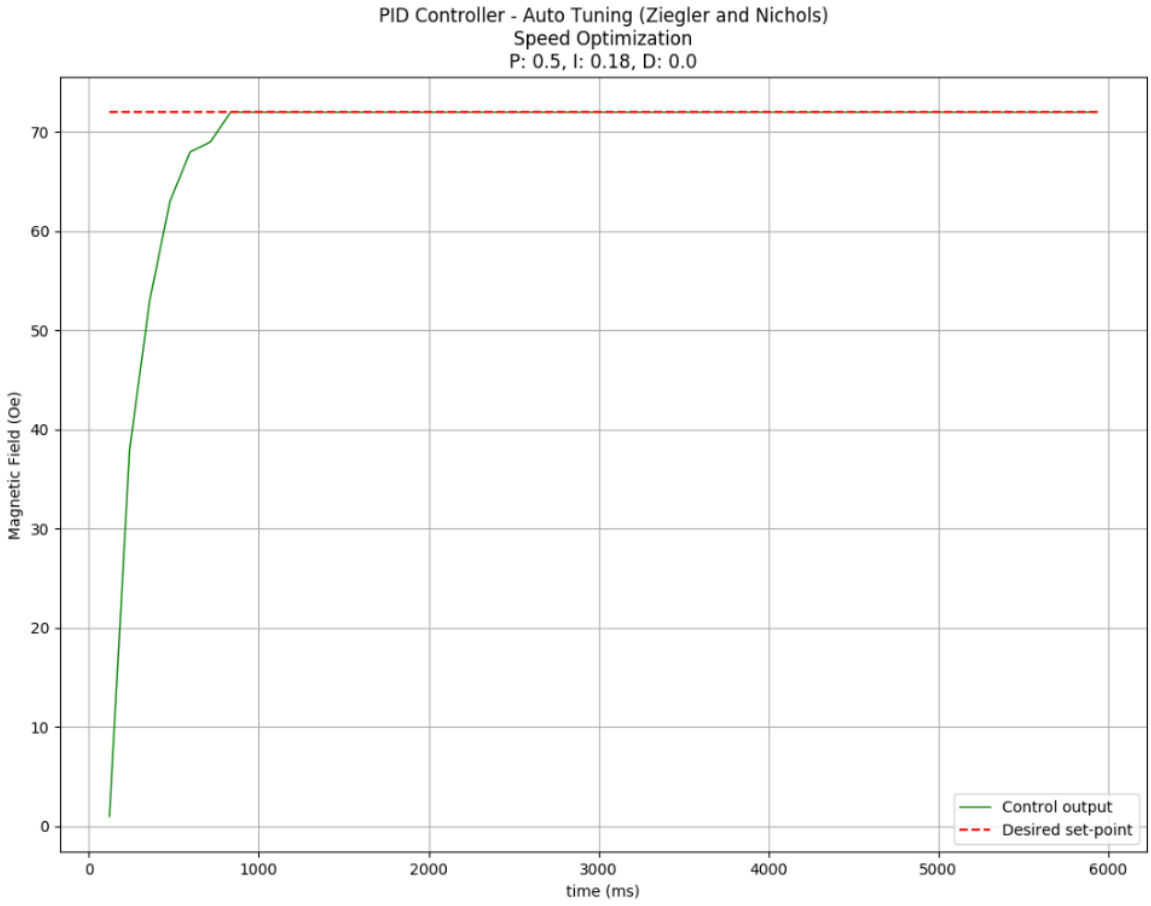


Figure 5.51: Speed optimized auto tuning (Ziegler-Nichols method) of the intelligent controller based on a Raspberry Pi. PI type of controller with pre-set gains Kp = 0.5 and Ki = 0.18. Desired magnetic field is 72 Oe and starting magnetic field is 0 Oe. Initial error is 72 Oe, reached magnetic field is 72 Oe and settling time is 800 ms. This tuning configuration fits-for-purpose of the intelligent controller based on a Raspberry Pi and is faster than the current optimal Ziegler-Nichols PI (Kp = 0.45 and Ki = 0.27) tuning configuration.

Table 5.2 summarizes the speed optimized Ziegler-Nichols' auto tuning tests for the intelligent controller based on a Raspberry Pi.

No.	Type of controller	Desired MF (Oe)	Starting MF (Oe)	Initial Error	Reached MF (Oe)	Settling time (ms)
1	PI	-120	0	-120	-120	800
2	PI	120	0	120	120	950
3	PI	-705	845	-1550	-705	1400
4	PI	845	-704	1549	845	1600
5	PI	-72	0	-72	-72	800
6	PI	72	0	72	72	800

Table 5.2: Speed optimization: Ziegler-Nichols auto tuning method - Tests log for the intelligent controller based on a Raspberry Pi. PI gains are set to Kp = 0.5 and Ki = 0.18.

Speed optimized Ziegler-Nichols' auto tuning tests outcome a faster PI controller based on a Raspberry Pi, comparing to the current optimal Ziegler-Nichols (Kp = 0.45 and Ki = 0.27) tuning configuration. In particular, speed optimized PI (Proportional-Integral) intelligent controller takes only ~ 800 *ms*, while previously it required ~ 1000 *ms* to settle. For a very large changes, where there is a big difference between the desired and the starting magnetic fields, speed optimized PI (Proportional-Integral) intelligent controller responses in just ~ 1500 *ms*, while previously it required ~ 2000 *ms* in order to settle.

Finally, it is important that the speed optimized PI (Proportional-Integral) intelligent controller enhances the system's response time, without "trading-off" any other performance indices. However, as it can be observed through Figure 5.49, signal becomes slightly noisier when a very large change (from -704 Oe to 845 Oe) is demanded in the magnetic field.

Chapter 6

CONCLUSIONS AND FUTURE WORK

In this chapter, we will draw the final conclusions (see section 6.1) of the project and identify further improvements and directions as a future work (see section 6.2).

6.1 Conclusions

Some of the latest control system advancements include the development of intelligent micro-machines between 1988 and 2003 [8]. Recently, there has been a growing interest in intelligent controllers which are widely used in the industrial society. In the form of digital microprocessors, automated intelligent controllers have been shown a significant enhancement of products quality, increase of productivity, improvement of contingency plans and reduction of labor costs [5, 7].

In this project, an intelligent controller based on a Raspberry Pi is proposed, which deploys the PID (Proportional-Integral-Derivative) logic. As a part of multi-component system, intelligent controller is capable of controlling and maintaining the field of an electromagnet by doing all the required calibrations and processing, including the adjustment of the electrical current supplied to the coils of the electromagnet. Desire magnetic field is being entered by the user through a PC.

The design and implementation of such closed-loop (or feedback) control system, where the intelligent controller is integrated into a standalone instrument, as a part of multi-component system was a complex and challenging procedure. Other hardware instruments of the system include the gaussmeter (with Hall sensor), the power supply, the electromagnet and the host PC. It is important that different hardware choices have

been reviewed and evaluated prior to the selection of the digital microprocessor for the intelligent controller. The Raspberry Pi 3 Model B+, a compact computer board, has been chosen due to its sufficient number of Universal Serial Bus (USB) ports, familiarity with the programming language offered (Python), cost-effectiveness and portability, together with remarkable performance, power-consumption and storage.

For the successful execution and monitoring of the project, a set of objectives have been identified which were then followed as different tasks/activities. First task included the design of an aluminium probe stand which was used to ensure that the probe (or Hall sensor) is held exactly in place for accurate field measurements. The probe stand consists of two different units (bottom and top) with a semi-hole on each, acting as the probe holders, responsible for holding the Hall sensor in place (centered between the magnets of the electromagnet). A base with two M6 drilled and tapped holes for mounting the probe holders, together with a secured place for adjusting the electromagnet is also included.

After assembling the hardware components of the proposed magnetic field system, it was time to design and implement the software. Initial step of the implementation phase constituted of the establishment of individual communications between each hardware instrument of the system with the intelligent controller. Power supply and Raspberry Pi communication was one of the most time-consuming technical task of the project as GPIB drivers had to be manually installed and configured on Raspberry Pi's Linux-based operating system (OS).

Furthermore, after setting-up the environment, controlled process had to be mathematically modelled using look-up tables technique. This is because the magnetic field input/output is a result of the electrical current flowing in the coils of the electromagnet. Therefore, correlation between the controlled variable (electrical current) and the input/output (magnetic field) of the proposed feedback control system had to be obtained by matching controlled variable-to-input/output correspondents.

Moreover, a custom-initiated software framework has been developed using Python programming language in order to control the magnetic field from an electromagnet, through the intelligent controller. Software comprises of *SerialInterface*, *GpibInterface*, *LakeShore425_Gauss*, *KepcoBOP*, *PID*, *PIDTuning*, *MagnetCalibration* and *UI* Python classes. Each class denotes an object which can be instantiated in the intelligent controller main Python script. Object of a class can also be instantiated in another class, where relationship between the two classes is pre-defined. Considering the importance of software re-usability, object-oriented Python classes designed in a way that

they can be utilized by an upgraded version of this project or any other project aiming to communicate with the specified instruments or interfaces.

An important feature of the software, enhancing the quality of the proposed solution, is the magnet calibration which automates the look-up tables technique, and thus, handles any other magnet than the specified one for this project. Calibration feature would also ensure the accuracy level of the system if the probe (or Hall sensor) is changed. For instance, this project uses a high sensitivity probe (HSE) and an electromagnet with electrical current limits of +-3 Amps. If for example, the electromagnet remains the same in the system, but the probe (or Hall sensor) is changed to a high stability (HST) version, correlation values are expected to be slightly different even with the same electrical current limits of +-3 Amps. Therefore, automated magnet calibration feature would ensure future compatibility of the software with any other magnet, as well as, any other probe (or Hall sensor) version.

In addition, usability of the proposed intelligent controller based on a Raspberry Pi system had to be considered closely, due to the user interaction with the program. Therefore, both terminal-based and graphical UIs have been implemented.

Final software integration was handled through the intelligent controller main Python script. This script accommodates the PID logic in order to dynamically control and maintain the desired output (magnetic field) of the system using feedback signal, until the error signal has been eliminated.

After the implementation of the system, testing and evaluation were crucial in order to remark, and thus, guarantee the coherent behaviour of the automated control solution. Different hardware and software units tests have been performed, together with integrated system test. For the integrated system testing, different manual and auto tuning methods have been investigated. The evaluation of the shortlisted tuning configurations showed that a PI (Proportional-Integral) type of controller is the most suitable for this system. It is important that both P-only and PID (Proportional-Integral-Derivative) type of controllers have been discarded, due to steady-state error (”or droop”) and high-frequency noise outcomes.

Results denoted that the prompted desire magnetic fields are successfully reached using the intelligent PI (Proportional-Integral) controller based on a Raspberry Pi. In terms of the system performance, magnetic field controlled process is stable, accurate and fast, without any signs of either overshooting or undershooting. Speed optimization is also undertaken, enhancing the response time of the system even more.

In conclusion, the proposed magnetic field control system is fully functional, and

thus, the project met its original aim/goal, together with the set objectives (see section 1.1).

6.2 Future Work

Different ways of improving the proposed magnetic field control system have been identified, including further software development (see section 6.2.1), research and implementation of different control techniques (see section 6.2.2), together with the integration into a broader system (see section 6.2.3).

6.2.1 Further Software Development

There are two recommended software advancements that would enhance the overall functionality, as well as, the usability of the proposed magnetic field control system:

1. At the moment, automated magnet calibration feature is only taking the electrical current limit as an input parameter. It would be desirable if the *MagnetCalibration* Python class is further developed in order to accommodate the precision level of the increment/decrement steps of the electrical current as an input parameter as well. Therefore, end-user would be able to select the desire precision level for the automated magnet calibration procedure. For this work, increment/decrement steps of the electrical current are pre-set to 0.25 Amps.

2. Except from the terminal-based UI, a simple Graphical User Interface (GUI) has also been developed in this revision of the software, using Tkinter Python library. GUI consists of main and pop-up windows which support the principle of entering the desired magnetic field and waiting for the output response in order to be displayed. However, a better approach would update the end-user in real-time, while the control output is being processed and maintained. One possible way of doing that would be through the usage of multiple threads in the program. Further features of magnet calibration and plotting can also be added to the GUI.

6.2.2 Research and Development of Different Control Techniques

Although the proposed intelligent controller which utilizes the PID logic is fully operational and fit-for-purpose of the project, other techniques might be investigated and

developed. This is because PID control process is based on linear modelling and in reality, at some point, every system might outcome a non-linear behaviour under different circumstances. For this system, non-linear behaviour is most likely to be exhibited when the magnetic field is ranging at low levels (e.g. between -70 Oe and 70 0e).

Modifications to the original PID algorithm or the implementation of other control techniques for the purposes of the magnetic field control system would include the PID error-squared (or gain scheduling) [16], feed-forward [32] and fuzzy logic [33]. It is important that the current version of the software is designed by following the object-oriented principles. Therefore, any modifications or additions to the existing software should be straightforward.

6.2.3 Integration into a Broader System

The proposed intelligent controller based on a Raspberry Pi has been successfully integrated into a standalone instrument, as a part of multi-component system. However, in the future it would be desirable if the intelligent controller can be further integrated into a broader system, where in addition to the magnetic field, other variables would also be controlled (e.g. room temperature).

As a pre-requisite, for a thorough integration into a wider system, intelligent controller should be able to connect with the host PC using a USB-to-USB wired connection. This is because Universal Serial Bus (USB) is the most reliable, fast and commonly-used interface in the instrumentation world, comparing to the GPIB and Ethernet (or LAN) interfaces. In particular, General Purpose Interface Bus (GPIB) is constantly being replaced by other modern interfaces (e.g. USB and Ethernet), while Ethernet (or LAN) communication might not be practical as PCs usually have only one Ethernet socket which is primarily used for network purposes.

For the proposed magnetic field control system, intelligent controller based on a Raspberry Pi has been connected to the host PC using an Ethernet-to-Ethernet connection. Communication between the intelligent controller based on a Raspberry Pi and the PC was then established using VNC Viewer, a third-party virtual tool. This action was mandatory as Raspberry Pi has to be booted on its own operating system (Linux-based), and thus, a direct USB-to-USB connection with the host PC is not applicable.

Alternative ways of including USB connection to at least one end, would include the utilization of an Ethernet-to-USB adaptor or directly connect TTL (female connectors) to the Raspberry Pi's GPIO male pins. In fact, USB-to-TTL connection would boot the Raspberry Pi into a Windows PC. However, a further limitation applied here

as such connection would only be able to provide a command line interface through an SSH software tool (e.g. PuTTY). Therefore, the usage of a Graphical User Interface (GUI), as the one proposed in this system, would never be achievable through a USB-to-TTL connection between the PC and the Raspberry Pi.

Conclusively, the proposed intelligent controller based on a Raspberry Pi would not be suitable for a future integration into a broader system. This is due to Raspberry Pi's limitation of requiring to boot its own Linux-based operating system, and thus, a direct USB-to-USB communication cannot be supported. Considering the above limitation of Raspberry Pi's, for a future integration into a wider system, alternative hardware choices (in the form of digital microcontrollers) for the intelligent controller should be reviewed and selected, appropriately. It is important that the hardware choice for the intelligent controller should support Windows environment, considering that this is the most common operating system in the instrumentation world. A good starting point of investigating different hardware choices for the intelligent controller would be the EVB-USB5926 Evaluation board, offered by Microchip [70].

REFERENCES

[1] William Bolton. *Instrumentation and Control Systems*, pages 1–13; 81–118; 195–209; 281–301. Elsevier, Oxford, UK, 2nd edition, 2015.

[2] Borja Bordel Sanchez, Ramon Alcarria, Diego Sanchez de Rivera, and Alvaro Sanchez-Picot. Enhancing process control in industry 4.0 scenarios using cyber-physical systems. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 7:41–64, 2016.

[3] Andreja Rojko. Industry 4.0 concept: Background and overview. *International Journal of Interactive Mobile Technologies (iJIM)*, 11:77–90, 2017. doi:10.3991/ijim.v11i5.7072.

[4] Stuart Bennett. A brief history of automatic control. *IEEE Control Systems Magazine*, 16(3):17–25, 1996. doi:10.1109/37.506394.

[5] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*, pages 1–36. Pearson, 13th edition, 2017.

[6] Ajit K. Mandal. *Introduction to Control Engineering: Modeling, Analysis and Design*, pages 1–20. New Age International, New Delhi, India, 2006.

[7] K.L.S. Sharma. *Overview of Industrial Process Automation*, pages 1–14. Elsevier, 2nd edition, 2016.

[8] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*, pages 1–34. Pearson, 12th edition, 2010.

[9] Melanie Arntz, Terry Gregory, and Ulrich Zierahn. The risk of automation for jobs in oecd countries. *OECD Social, Employment and Migration Working Papers*, (189):1–35, 2016. doi: `https://doi.org/10.1787/5jlz9h56dvq7-en`.

[10] Frank Lamb. *Industrial Automation: Hands On*, pages 1–6. McGraw-Hill Professional, USA, 2013. doi:10.1036/9780071816472.

[11] Richard Bellman. Control theory. *Scientific American*, 211:186–200, 1964. doi:10.1038/scientificamerican0964-186.

[12] Katsuhiko Ogata. *Modern control engineering*, pages 1–63. Pearson, Boston, MA, 5th edition, 2010.

[13] Tarek Abdelzaher, Yixin Diao, Joseph L. Hellerstein, Chenyang Lu, and Xiaoyun Zhu. *Introduction to Control Theory And Its Application to Computing Systems*, pages 185–215. Springer US, Boston, MA, 2008. doi: https://doi.org/10.1007/978-0-387-79361-0_7.

[14] Thomas Dunn. *Basics of Control Systems (Chapter 10)*, pages 106–110. William Andrew Publishing, Oxford, 2015. doi: https://doi.org/10.1016/B978-0-323-26436-5.00010-2.

[15] Naser Mehrabi and John McPhee. *Model-Based Control of Biomechatronic Systems (Chapter 4)*, pages 95–108. Academic Press, 2019. doi: https://doi.org/10.1016/B978-0-12-812539-7.00004-0.

[16] Karl J. Astrom and Tore Hagglund. *PID Controllers: Theory, Design and Tuning*, pages 59–120. ISA, Research Triangle Park, NC, USA, 2nd edition, 1995.

[17] Shrikrishna N. Joshi. *Module 3: Programmable Logic Devices (PLDs) - Lecture 4: PID controllers*. National Programme on Technology Enhanced Learning (NPTEL), 2013. [Online]. Available at: https://nptel.ac.in/courses/112103174/module3/lec4/1.html [Accessed: 25 June 2019].

[18] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*, pages 17–23. Princeton University Press, Oxfordshire, UK, 2nd edition, 2008.

[19] Philipp K. Janert. *Feedback Control for Computer Systems*, pages 19–21; 94–96. O'Reilly Media, Inc., Sebastopol, CA, USA, 1st edition, 2013.

[20] John M. Hughes. *Real World Instrumentation with Python: Automated Data Acquisition and Control Systems*, pages 303–348. O'Reilly Media, Inc., 1st edition, 2010.

[21] S. Simrock. Control theory. *CERN, DESY, Hamburg, Germany*, pages 73–130, 2008. doi:10.5170/CERN-2008-003.73.

[22] Panus J. Antsaklis and Zhiqiany Gao. *Control Systems*, pages 19.1–19.30. The Electronics Engineers' Handbook, McGraw-Hill, Section 19, 5th edition, 2005.

[23] William Dunn. *Fundamentals of Industrial Instrumentation and Process Control*, pages 2–6; 241–257. McGraw-Hill Professional, US, 2005. doi:10.1036/0071457356.

[24] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005. doi:10.1109/TCST.2005.847331.

[25] Karl J. Astrom and Tore Hagglund. *Advanced PID control*, pages 1–11. ISA, 2006.

[26] Karl J. Astrom and Tore Hagglund. The future of pid control. *Control Engineering Practice*, 9(11):1163–1175, 2001. doi: `https://doi.org/10.1016/S0967-0661(01)00062-4`.

[27] O. Arrieta and R. Vilanova. Servo/regulation tradeoff tuning of pid controllers with a robustness consideration. In *46th IEEE Conference on Decision and Control*, pages 1838–1843, 2007. doi:10.1109/CDC.2007.4434930.

[28] J.G. Ziegler and N.B. Nichols. Optimum settings for automatic controllers. *Transactions of the ASME*, 64(8):759–768, 1993. doi:10.1115/1.2899060.

[29] Michael L. Luyben and William L. Luyben. *Essentials of Process Control*, pages 92–99. McGraw-Hill, Singapore, 1997.

[30] Mohammad Shahrokhi and Alireza Zomorrodi. Comparison of pid controller tuning methods. *Department of Chemical & Petroleum Engineering, Sharif University of Technology*, pages 1–12, no date.

[31] Yun Li, Kiam Heong Ang, and G. Chong. Pid control system analysis and design: Problems, remedies, and future directions. *IEEE Control Systems Magazine*, 26(1):32–41, 2006.

[32] Wolfgang Altmann, David Macdonald, and Steve Mackay, editors. *Concepts and applications of feedforward control (chapter 10)*, pages 142–146. Newnes, Oxford, 2005. doi: https://doi.org/10.1016/B978-075066400-4/50010-0.

[33] Jian-Xin Xu, Chang-Chieh Hang, and Chen Liu. Parallel structure and tuning of a fuzzy pid controller. *Automatica*, 36(5):673–684, 2000. doi: https://doi.org/10.1016/S0005-1098(99)00192-2.

[34] Michael J. Pont. *Patterns for Time-triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*, pages 1–25. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2001.

[35] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, pages 87–101, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[36] Fabian Scheler and Wolfgang Schroeder-Preikschat. Time-triggered vs. event-triggered: A matter of configuration? In *ITG FA 6.2 Workshop on Model-Based Testing, GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 13th GI/ITG Conference Measuring, Modelling, and Evaluation of Computer and Communications*, pages 1–6, 2006.

[37] Lake Shore Cryotronics. *Users Manual Model 425 Gaussmeter Rev. 1.2*. Lake Shore Cryotronics, 2017. pages 1-66. [Online]. Available at: https://www.lakeshore.com/docs/default-source/product-downloads/manuals/425manual.pdf?sfvrsn=88ad25ad_4 [Accessed: 19 April 2019].

[38] Lake Shore Cryotronics. *Hall Probe Stands*. Lake Shore Cryotronics, 2017. [Online]. Available at: https://shop.lakeshore.com/magnetic-products/hall-probes/400-series-probes/hall-probe-accessories/hall-probe-stands.html [Accessed: 20 April 2019].

[39] Kepco. *Operators Manual - BOP (M) (D) 100W, 200W, 400W BIPOLAR POWER SUPPLY*. Kepco, 2011. pages 1-70. [Online]. Available at: http://www.mmr-tech.com/PDFs/BOP_Operating_Manual.pdf [Accessed: 18 April 2019].

[40] Kepco. *Operators Manual - BIT 4886 DIGITAL INTERFACE CARD*. Kepco, 2013. pages 1-92. [Online]. Available at: https://www.kepcopower.com/support/bit4886opr-19.pdf [Accessed: 18 April 2019].

[41] National Instruments. *GPIB-USB-HS - GPIB Instrument Control Device*. National Instruments, 2019. [Online]. Available at: https://www.ni.com/en-gb/support/model.gpib-usb-hs.html [Accessed: 25 April 2019].

[42] National Instruments. *INSTALLATION GUIDE AND SPECIFICATIONS - GPIB Hardware*. National Instruments, 2015. pages 34-35. [Online]. Available at: http://www.ni.com/pdf/manuals/370426r.pdf [Accessed: 25 April 2019].

[43] C.R. Nave. *Hyper Physics-Electricity and Magnetism: Electromagnet*. Georgia State University, 2016. Georgia State University. [Online]. Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/elemag.html [Accessed: 18 April 2019].

[44] Arduino. *Arduino Mega 2560 Rev3*. Arduino, 2019. [Online]. Available at: https://store.arduino.cc/mega-2560-r3 [Accessed: 15 February 2019].

[45] RS Components. *Raspberry Pi Model Comparison Table*. RS Components, no date. page 1. [Online]. Available at: https://docs-emea.rs-online.com/webdocs/1662/0900766b8166246a.pdf [Accessed: 16 February 2019].

[46] RS Components. *Raspberry Pi 3 Model B+*. RS Components, no date. [Online]. Available at: https://uk.rs-online.com/web/p/processor-microcontroller-development-kits/1373331/ [Accessed: 16 February 2019].

[47] Raspberry Pi Foundation. *Raspberry Pi 3 Model B+*. Raspberry Pi Foundation, no date. [Online]. Available at: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ [Accessed: 16 February 2019].

[48] Raspberry Pi Foundation. *Raspberry Pi 3 Model B+ Product Brief*. Raspberry Pi Foundation, no date. pages 1-5. [Online]. Available at: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf [Accessed: 16 February 2019].

[49] E. Upton and G. Halfacree. *Raspberry Pi User Guide*. John Wiley & Sons, 4th edition, 2016. pages 153-179. [Online]. Available at: https://ebookcentral.proquest.com/lib/manchester/detail.action?docID=4635121 [Accessed: 18 February 2019].

[50] R. Roy and V. Bommakanti. *Odroid-XU4 User Manual*. HardKernel, 2015. pages 1-87. [Online]. Available at: https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf [Accessed: 19 February 2019].

[51] Odroid. *Odroid-XU4*. HardKernel, 2019. [Online]. Available at: https://www.odroid.co.uk/hardkernel-odroid-xu4/odroid-xu4 [Accessed: 19 February 2019].

[52] Aivar Annamaa. Introducing thonny, a python ide for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 117–121, New York, NY, USA, 2015. ACM. doi:10.1145/2828959.2828969.

[53] Thonny. *Thonny Python IDE for beginners*. Thonny, 2019. [Online]. Available at: https://thonny.org/ [Accessed: 10 June 2019].

[54] GitHub. *GitHub Version Control Tool*. GitHub Inc., 2019. [Online]. Available at: https://github.com/ [Accessed: 10 June 2019].

[55] RealVNC. *Download VNC Viewer*. RealVNC, 2019. [Online]. Available at: https://www.realvnc.com/en/connect/download/viewer/ [Accessed: 10 June 2019].

[56] RealVNC. *Raspberry Pi*. RealVNC, 2019. [Online]. Available at: https://www.realvnc.com/en/raspberrypi/ [Accessed: 10 June 2019].

[57] Lake Shore Cryotronics. *Instrument Communication Utility*. Lake Shore Cryotronics, 2019. [Online]. Available at: https://www.lakeshore.com/resources/software/instrument-communication-utility [Accessed: 10 June 2019].

[58] National Instruments. *National Instruments VISA*. National Instruments, 2014. [Online]. Available at: https://www.ni.com/visa/ [Accessed: 10 June 2019].

[59] National Instruments. *NI-VISA Downloads*. National Instruments, 2019. [Online]. Available at: https://www.ni.com/en-gb/support/downloads/drivers/download.ni-visa.html#305862 [Accessed: 10 June 2019].

[60] Mohammad Dabbagh, Sai Peck Lee, and Reza Parizi. Functional and non-functional requirements prioritization: empirical evaluation of ipa, ahp-based, and ham-based approaches. *Soft Computing*, 20:4497–4520, 2015. doi:10.1007/s00500-015-1760-z.

[61] S. Lauesen. Usability requirements in a tender process. In *Proceedings 1998 Australasian Computer Human Interaction Conference. OzCHI'98 (Cat. No.98EX234)*, pages 114–121, 1998. 10.1109/OZCHI.1998.732203.

[62] Eduardo Miranda. Time boxing planning: buffered moscow rules. *ACM SIGSOFT Software Engineering Notes*, 36:1–5, 2011. doi:10.1145/2047414.2047428.

[63] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, 168(1):70–118, 2005. doi: `https://doi.org/10.1016/j.artint.2005.05.003`.

[64] Giorgos Tsapparellas. *Intelligent controller based on a Raspberry Pi*. GitHub Inc., 2019. [Online]. Available at: `https://github.com/GTsapparellas/Intelligent-controller-based-on-a-Raspberry-Pi` [Accessed: 10 August 2019].

[65] National Instruments. *NI-488.2*. National Instruments, 2019. [Online]. Available at: `http://www.ni.com/en-gb/support/downloads/drivers/download.ni-488-2.html#305442` [Accessed: 20 June 2019].

[66] Illya Tsemenko. *Setting up and using NI USB-GPIB-HS in Raspberry Pi Linux*. xDevs.com, 2015. [Online]. Available at: `https://xdevs.com/guide/ni_gpib_rpi/` [Accessed: 21 June 2019].

[67] Dave Penkler and Frank Hess. *Linux GPIB Support*. Source Forge, 2019. [Online]. Available at: `https://sourceforge.net/projects/linux-gpib/` [Accessed: 25 June 2019].

[68] Milo Petrovi. *Correlation and regression calculator*. Mathportal, no date. [Online]. Available at: `https://www.mathportal.org/calculators/statistics-calculator/correlation-and-regression-calculator.php` [Accessed: 10 July 2019].

[69] Alan Dennis, Barbara H. Wixom, and David Tegarden. *System Analysis Design UML Version 2.0: An Object-Oriented Approach*, pages 414–421. John Wiley & Sons, Inc., USA, 4th edition, 2012.

[70] Microchip Technology Inc. *EVB-USB5926 Evaluation Board*. Microchip Technology Inc., 2019. [Online]. Available at: `https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/EVB-USB5926` [Accessed: 25 August 2019].

Appendix A

Probe Stand Design Schematics

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

31st May 2019

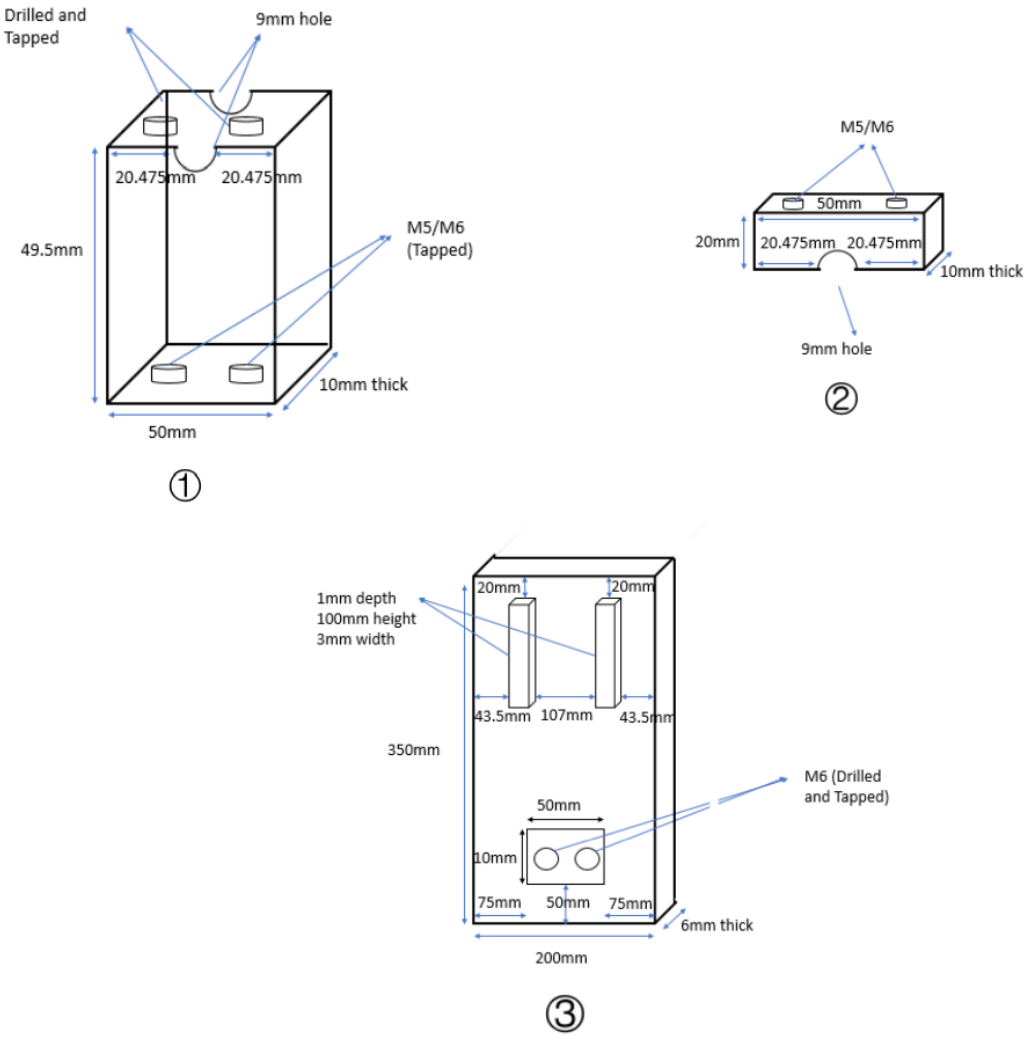


Figure A.1: Design schematics of the custom-initiated aluminium probe stand. Bottom probe holder is marked with 1, top probe holder with 2 and base with 3.

Appendix B

List of Hardware Instruments

Version

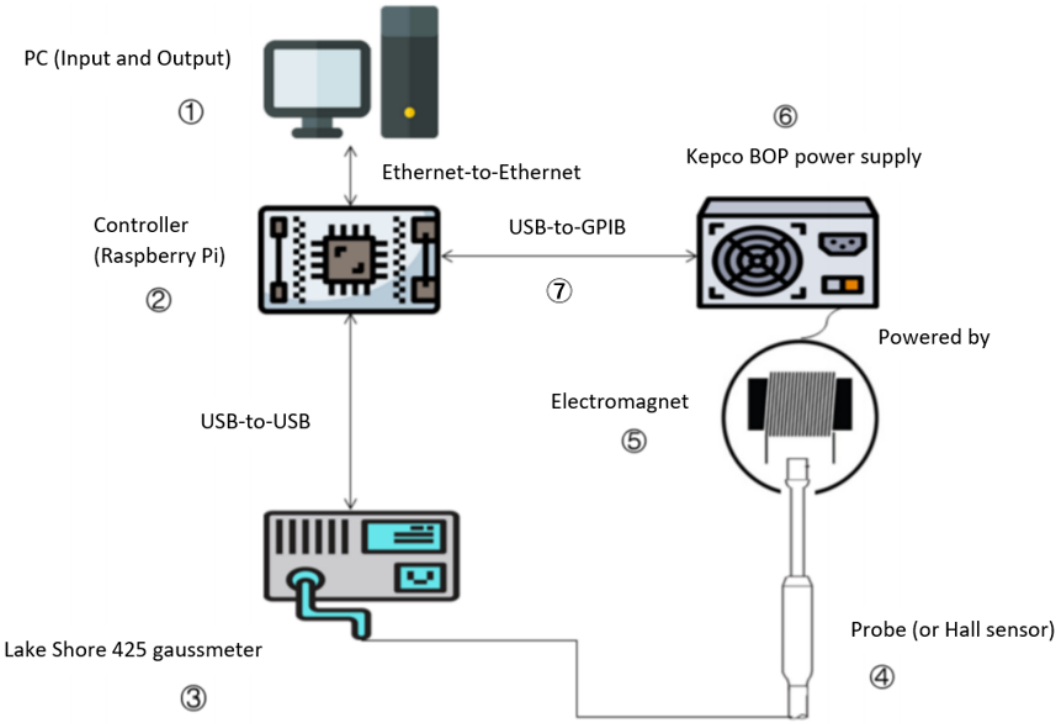
1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

5th June 2019



No.	Hardware Instrument	Model	Manufacturer
1	PC	481-009-600 REV.M	Bruker
2	Raspberry Pi	3 B+	Raspberry Pi Foundation
3	Gaussmeter	425	Lake Shore Cryotronics
4	Probe (or Hall Sensor)	High-sensitivity (HSE)	Lake Shore Cryotronics
5	Electromagnet	-	Built in UoM
6	Power Supply	BOP 20-20ML	Kepco
7	GPIB-to-USB control device	HS	National Instruments

Table B.1: Full list of hardware instrumentation.

Appendix C

List of Software and Tools

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

12th June 2019

No.	Software/Tool	Version	Support OS	Provider	Comments
1	Python	2.7	Windows, Linux, Mac OS X and Other	Python Software Foundation	Open-source
2	Thonny IDE	3.1.2	Windows, Mac and Linux	Aivar Annamaa and contributors	Open-source
3	GitHub	-	Any platform through web connection	Microsoft	Open-source
4	VNC Viewer	6.4.1	Windows, MacOS, Linux, Raspberry Pi and Other	RealVNC	Open-source
5	Lake Shore Communication	1.1	Windows Vista and newer	Lake Shore Cryotronics, Inc.	Open-source
6	NI VISA Interactive Control	17.0	Windows 7, 8.1 and 10	National Instruments	Open-source

Table C.1: Full list of software and tools.

Appendix D

List of System Requirements

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

15th June 2019

No.	System Requirement	Type	”MoSCoW” Prioritization
1	The PC must communicate with the intelligent controller.	Functional	Must have
2	Communication between the PC and the intelligent controller must be established using appropriate end-to-end wired connection (e.g. Ethernet or USB) along with a third-party software, if required.	Non-functional	Must have
3	The Gaussmeter (including Hall sensor) must communicate with the intelligent controller.	Functional	Must have
4	Communication between the Gaussmeter (including Hall sensor) and the intelligent controller must be established using USB end-to-end wired connection along with serial interface software.	Non-functional	Must have
5	The Power supply must communicate with the intelligent controller.	Functional	Must have
6	Communication between the Power supply and the intelligent controller must be established using appropriate end-to-end wired connection (e.g. GPIB-USB or RS232) along with gpib or serial interface software.	Non-functional	Must have
7	The system must gather desired magnetic field as input.	Functional	Must have
8	The desired magnetic field input must be entered by the user through a UI.	Non-functional	Must have
9	The system must monitor the actual magnetic field.	Functional	Must have

10	Actual magnetic field must be monitored remotely using gaussmeter, Hall sensor, intelligent controller and related queries/commands.	Non-functional	Must have
11	The system must amend the electrical current going through the electromagnet, and thus, modifying the actual magnetic field as desired.	Functional	Must have
12	The electrical current must be amended remotely using power supply, intelligent controller and SCPI messages.	Non-functional	Must have
13	The system must make the actual magnetic field output available to the user.	Functional	Must have
14	The actual magnetic field output must be made available to the user using a UI.	Non-functional	Must have
15	The system must provide a coherent UI to the user, conforming ease of use and readability	Usability	Must have
16	The system must incorporate the PID controller logic.	Functional	Must have
17	PID controller should be tuned using manual and auto tuning.	Non-functional	Should have
18	The system should provide a plotting feature for analyzing performance indices of the control procedure.	Functional	Should have
19	The system should provide stable, accurate and fast-paced output, without any overshooting.	Functional	Should have
20	The system could provide a magnet calibration feature.	Functional	Could have
21	The system could provide a GUI, for an enhanced user experience.	Usability	Could have

Table D.1: Full list of system requirements classified and prioritized using ”MoSCoW” method.

Appendix E

List of Functions - LakeShore425_Gauss Python Class

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

10th July 2019

No.	Function	Command/Query
1	q_ProcessLastQueryReceived	?
2	q_ID	*IDN?
3	c_Reset	*RST
4	c_InputAlarmParameter	ALARM
5	q_InputAlarmParameter	ALARM?
6	q_AlarmStatus	ALARMST?
7	c_AutoRange	AUTO
8	q_AutoRange	AUTO?
9	c_AlarmAudible	BEEP
10	q_AlarmAudible	BEEP?
11	c_DisplayContrast	BRIGT
12	q_DisplayContrast	BRIGT?
13	c_FactoryDefaults	DFLT 99
14	q_KeypadStatus	KEYST?
15	c_FrontPanelKeypadLock	LOCK
16	q_FrontPanelKeypadLock	LOCK?
17	c_MaxHold	MXHOLD
18	q_MaxHold	MXHOLD?
19	c_MaxHoldReset	MXRST
20	q_OperationalStatus	OPST?
21	c_ProbeFieldCompensation	PRBFCOMP
22	q_ProbeFieldCompensation	PRBFCOMP?
23	q_ProbeSensitivity	PRBSENS?
24	q_ProbeSerialNumber	PRBSNUM?
25	c_FieldRange	RANGE
26	q_FieldRange	RANGE?
27	q_FieldReading	RDGFIELD?
28	c_MeasurementMode	RDGMODE
29	q_MeasurementMode	RDGMODE?
30	q_MaxMinReading	RDGMNMX?
31	q_MaxReading	RDGMX?
32	q_RelativeReading	RDGREL?

33	c_RelativeMode	REL
34	q_RelativeMode	REL?
35	c_RelayControlParameter	RELAY
36	q_RelayControlParameter	RELAY?
37	q_RelayStatus	RELAYST?
38	c_RelativeSetpoint	RELSP
39	q_RelativeSetpoint	RELSP?
40	q_ProbeType	TYPE?
41	c_FieldUnits	UNIT
42	q_FieldUnits	UNIT?
43	c_ClearZeroProbe	ZCLEAR
44	q_ZeroProbe	ZPROBE
45	c_testGaussCommand	ANY Command
46	q_testGaussQuery	ANY Query

Table E.1: All the available functions of the custom-initiated *LakeShore425_Gauss* Python class, each forming a dedicated message string (command or query). Pre-fix c_ indicates a function returning a command message string, while q_ pre-fix indicates a function returning a query message string.

Appendix F

List of Functions - KepcoBOP Python Class

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

25th July 2019

No.	Function	SCPI Command/Query
1	c_ClearStatus	*CLS
2	c_EventStatusEnable	*ESE
3	q_EventStatusEnable	*ESE?
4	q_EventStatusRegister	*ESR?
5	q_ID	*IDN?
6	c_OperationComplete	*OPC
7	q_OperationComplete	*OPC?
8	q_Options	*OPT?
9	c_Recall	*RCL
10	c_Reset	*RST
11	c_Save	*SAV
12	c_ServiceRequestEnable	*SRE
13	q_ServiceRequestEnable	*SRE?
14	q_StatusByteRegister	*STB
15	c_Trigger	*TRG
16	q_SelfTest	*TST?
17	c_WaitToContinue	*WAI
18	c_CalibrateProtect	CAL:CPR
19	c_CalibrateStatus	CAL:STAT
20	q_CalibrateStatus	CAL:STAT?
21	c_CalibrateCurrent	CAL:CURR
22	c_CalibrateData	CAL:DATA
23	c_CalibrateDPOT	CAL:DPOT
24	c_CalibrateLowCurrent	CAL:LCURR
25	c_CalibrateLowVoltage	CAL:LVOLT
26	c_CalibrateSave	CAL:SAVE
27	c_CalibrateVoltage	CAL:VOLT
28	c_CalibrateVoltageProtect	CAL:VPR
29	c_CalibrateZero	CAL:ZERO
30	c_EnableSingleRegister	INIT:IMM
31	c_EDContinuousTriggers	INIT:CONT
32	q_EDContinuousTriggers	INIT:CONT?

33	q_MeasureCurrent	MEAS:CURR?
34	q_MeasureVoltage	MEAS:VOLT?
35	c_EDOutput	OUTP
36	q_EDOutput	OUTP?
37	c_OperatingMode	FUNC:MODE
38	q_OperatingMode	FUNC:MODE?
39	c_OperatingModeTrigger	FUNC:MODE:TRIG
40	q_OperatingModeTrigger	FUNC:MODE:TRIG?
41	c_ClearAllListEntries	LIST:CLE
42	c_ListCountExecution	LIST:COUN
43	q_ListCountExecution	LIST:COUN?
44	c_ListCountSkip	LIST:COUN:SKIP
45	q_ListCountSkip	LIST:COUN:SKIP?
46	c_AddCurrentToList	LIST:CURR
47	q_FindCurrentEntriesInList	LIST:CURR?
48	q_CurrentNumOfPointsInList	LIST:CURR:POIN?
49	c_ListDirectionExecution	LIST:DIR
50	q_ListDirectionExecution	LIST:DIR?
51	c_KeepDwellActiveFor	LIST:DWEL
52	q_ListDwellTimes	LIST:DWEL?
53	q_DwellTimesNumOf-PointsInList	LIST:DWEL:POIN?
54	c_ListOrderExecution	LIST:GEN
55	q_ListOrderExecution	LIST:GEN?
56	c_ListLocationToToBeQueried	LIST:QUER
57	q_ListLocationToBeQueried	LIST:QUER?
58	c_DwellTimeResolution	LIST:MODE
59	q_DwellTimeResolution	LIST:MODE?
60	c_ExecutionOrderFor-ListDataPoints	LIST:SEQ
61	q_ExecutionOrderFor-ListDataPoints	LIST:SEQ?
62	c_AddVoltageToList	LIST:VOLT
63	q_FindVoltageEntriesInList	LIST:VOLT?

64	q_Voltage	LIST:VOLT:POIN?
65	c_SetCurrentLevel	CURR
66	q_FindCurrentLevel	CURR?
67	c_SetCurrentMode	CURR:MODE
68	c_SetCurrentModeToTran	CURR:MODE TRAN
69	q_FindCurrentMode	CURR:MODE?
70	c_SetCurrentRange	CURR:RANG
71	q_FindCurrentRange	CURR:RANG?
72	c_SetAutoCurrentRange	CURR:RANG:AUTO
73	c_SetCurrentTriggerValue	CURR:TRIG
74	q_FindCurrentTriggerValue	CURR:TRIG?
75	c_SetVoltageLevel	VOLT
76	q_FindVoltageLevel	VOLT?
77	c_SetVoltageMode	VOLT:MODE
78	c_SetVoltageModeToTran	VOLT:MODE TRAN
79	q_FindVoltageMode	VOLT:MODE?
80	c_SetVoltageRange	VOLT:RANG
81	q_FindVoltageRange	VOLT:RANG?
82	c_SetAutoVoltageRange	VOLT:RANG:AUTO
83	c_SetVoltageTriggerValue	VOLT:TRIG
84	q_FindVoltageTriggerValue	VOLT:TRIG?
85	q_FindOperationCondition-RegisterValue	STAT:OPER:COND?
86	c_SetOperationEnableRegister	STAT:OPER:ENAB
87	q_FindOperationEnableRegister	STAT:OPER:ENAB?
88	c_FindChangesByOperaion-EnableRegister	STAT:OPER?
89	c_DisableReportingOf-StatusEvents	STAT:PRES
90	q_FindLatchConditionOf-QuestionableEventRegister	STAT:QUES?
91	q_FindQuestionableCondition-EnableRegister	STAT:QUES:COND?

92	c_SetQuestionableCondition-EnableRegister	STAT:QUES:ENAB
93	q_FindQuestionableCondition-EnableRegister	STAT:QUES:ENAB?
94	c_EmitBriefAudibleTone	SYST:BEEP
95	c_EDEchoMode	SYST:COMM:SER:ECHO
96	q_EDEchoMode	SYST:COMM:SER:ECHO?
97	c_EDDataFlowControl	SYST:COMM:SER:PACE
98	q_EDDataFlowControl	SYST:COMM:SER:PACE?
99	q_PostErrorMessages	SYST:ERR?
100	q_PostErrorCode	SYST:ERR:CODE?
101	q_PostAllErrorCodes	SYST:ERR:CODE:ALL?
102	c_SetPasswordEnableState	SYST:PASS:CEN
103	c_ClearPasswordEnableState	SYST:PASS:CDIS
104	c_SetNewPassword	SYST:PASS:NEW
105	q_FindPasswordState	SYST:PASS:STAT?
106	c_SetRemoteCommunication	SYST:REM
107	q_FindCommunicationMode	SYST:REM?
108	c_InitNVRAMVariables	SYST:SEC:IMM
109	c_SetSystemFunctions	SYST:SET
110	q_FindSetSystemFunctions	SYST:SET?
111	q_FindImplementedSCPIVersion	SYST:VERS?
112	sendMultipleKepcoSCPI	ANY SCPI Command(s)/Querie(s)
113	testKepcoSCPI	ANY SCPI Command/Query

Table F.1: All the available functions of the custom-initiated *KepcoBOP* Python class, each forming a dedicated SCPI message string (command or query). Pre-fix c_ indicates a function returning a command SCPI message string, while q_ pre-fix indicates a function returning a SCPI query message string.

Appendix G

Manual Tuning Tests Log

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

13th August 2019

No.	Type of controller	Desired MF (Oe)	Starting MF (Oe)	Initial Error	Reached MF (Oe)	Settling time (ms)
1	P-only	-385	0	-385	-370	5700
2	P-only	250	0	250	235	4200
3	PI	-100	334	-434	-225	3500
4	PI	255	0	255	270	5700
5	PI	-100	328	-428	-97	3800
6	PI	255	0	255	252	3000
7	PD	-100	0	-100	-97	4000
8	PD	100	0	100	100	5800
9	PID	-310	0	-310	-356	5000
10	PID	310	0	310	371	5000
11	PID	-310	0	-310	-303	5000
12	PID	310	0	310	306	5000

Table G.1: Manual tuning tests log for the intelligent controller based on a Raspberry Pi. Different Kp, Ki and Kd gains may be applied on the above P-only, PI, PD and PID type of controllers.

Appendix H

Auto Tuning Tests Log

Version

1.0 Original

Created By

Giorgos Tsapparellas

Date of Issue

14th August 2019

No.	Type of controller	Desired MF (Oe)	Starting MF (Oe)	Initial Error	Reached MF (Oe)	Settling time (ms)
1	P-only	-260	0	-260	-258	1700
2	P-only	750	0	750	744	1800
3	PI	-120	0	-120	-120	1200
4	PI	120	0	120	120	1000
5	PI	-705	845	-1550	-705	2000
6	PI	845	-705	1550	845	2000
7	PI	-72	0	-72	-72	1000
8	PI	72	0	72	72	1000
9	PID	-450	-200	-250	-448	2000
10	PID	450	0	450	448	2000

Table H.1: Ziegler-Nichols method - Auto tuning tests log for the intelligent controller based on a Raspberry Pi. Different Kp, Ki and Kd gains may be applied on the above P-only, PI and PID type of controllers.

No.	Type of controller	Desired MF (Oe)	Starting MF (Oe)	Initial Error	Reached MF (Oe)	Settling time (ms)
1	PI	-120	0	-120	-120	1800
2	PI	120	0	120	120	1800
3	PI	-705	842	-1547	-705	2800
4	PI	845	-704	1549	845	3100
5	PI	-72	0	-72	-72	1500
6	PI	72	0	72	72	1500
7	PID	-450	-202	-248	-458	2200
8	PID	450	0	450	448	2200

Table H.2: Tyreus-Luyben method - Auto tuning tests log for the intelligent controller based on a Raspberry Pi. Different Kp, Ki and Kd gains may be applied on the above PI and PID type of controllers.