UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

ENHANCING STUDENT LEARNING IN TEMPERATURE SENSING AND DATA ACQUISITION WITH PYRODAQ:
A Python-Based Approach for Controlling National Instruments Data Acquisition Devices

TRABAJO FIN DE GRADO

Septiembre –2023

AUTORA: Judit Danso Llaquet

DIRECTORA: Julia Arias Rodriguez

“The skill I was learning was a crucial one,

the patience to read things I could not yet understand.”

Tara Westover

TABLE OF CONTENT

LIST OF FIGURES

LIST OF SNIPPETS

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

Abbreviation	Definition
DAQ	Data Acquisition Device
GUI	Graphical Interface User
LabVIEW	Laboratory Virtual Instrument Engineering Workbench
NI	National Instruments
RTD	Resistance Temperature Detector

ABSTRACT

The motivation for this project arises from the importance of temperature sensing and data acquisition across various fields, particularly in electronics and engineering. It is a fundamental aspect of modern engineering, with applications ranging from climate control to industrial processes, healthcare, and scientific research.

In engineering, National Instruments (NI) and Data Acquisition (DAQ) devices are notable tools, facilitating interaction with sensors and instruments. However, in educational settings, these devices often face accessibility issues due to complex and proprietary software like LabVIEW (Laboratory Virtual Instrument Engineering Workbench).

This project's motivation is to address these challenges by harnessing Python's versatility and power. Python is an open-source language known for its simplicity and extensive libraries, making it ideal for scientific and engineering applications, including interfacing with NI DAQ devices.

The project's culmination is a GUI-based Python application, aiming to provide a user-friendly interface closely integrated with NI DAQ systems. It empowers users in temperature sensing and data acquisition, making these processes accessible and efficient. This project enhances the usability and accessibility of essential tools for engineers, researchers, and students in electronics and engineering.

The primary project’s aim is to streamline the construction process by offering a straightforward application. It simplifies the user experience and allows users to explore hardware-software connections, addressing gaps in traditional approaches. Additionally, the project emphasizes key concepts in electronic instrumentation, focusing on temperature measurement. A student guide is created to assist with installation, usage, and optimization.

The project is divided into several sections. Chapter 2 outlines the methodology, including requirement gathering, implementation, testing, and user evaluation. Chapter 3 details the development of the temperature sensing circuit control application, covering its features, architecture, and user interaction. Chapter 4 addresses practical aspects, including circuit setup, calibration, and results analysis. In the concluding chapter, the

project summarizes objectives and key findings, highlighting the application's usability and potential for enhancing learning experiences. It acknowledges the learning journey in Python programming and GUI design, emphasizing the importance of adaptability.

Overall, the project achieved its primary research objectives, delivering an educational application (PyroDAQ) for temperature sensing and data acquisition. It provides a foundation for further exploration and customization in the field, emphasizing the value of open-source tools and hands-on experiences in electronic instrumentation.

RESUMEN

La motivación de este proyecto surge de la importancia de la medida de la temperatura y la adquisición de datos en diversos campos, especialmente en electrónica e ingeniería. Es un aspecto fundamental de la ingeniería moderna, con aplicaciones que van desde el control climático hasta procesos industriales, atención médica e investigación científica.

En la ingeniería, los dispositivos de National Instruments (NI) y la adquisición de datos (DAQ) son herramientas destacadas que facilitan la interacción con sensores e instrumentos. Sin embargo, en entornos educativos, estos dispositivos a menudo enfrentan problemas de accesibilidad debido al software complejo y propietario como lo es LabVIEW (Laboratory Virtual Instrument Engineering Workbench).

La motivación de este proyecto está en abordar estos desafíos aprovechando la versatilidad y potencia de Python. Python es un lenguaje de código abierto conocido por su simplicidad y sus amplias bibliotecas, lo que lo hace ideal para aplicaciones científicas e ingenieriles, incluida la interfaz con dispositivos NI DAQ.

La culminación del proyecto es una aplicación Python basada en una interfaz gráfica de usuario (GUI), con el objetivo de proporcionar una interfaz amigable para el usuario estrechamente integrada con los sistemas NI DAQ. Empodera a los usuarios en la medida de la temperatura y la adquisición de datos, haciendo que estos procesos sean accesibles y eficientes. Este proyecto mejora la usabilidad y accesibilidad de herramientas esenciales para ingenieros, investigadores y estudiantes en electrónica e ingeniería.

El objetivo principal del proyecto es simplificar el proceso de construcción mediante la oferta de una aplicación sencilla. Simplifica la experiencia del usuario y permite a los usuarios explorar las conexiones entre hardware y software, abordando las deficiencias de enfoques tradicionales. Además, el proyecto enfatiza conceptos clave en la instrumentación electrónica, centrándose en la medida de la temperatura. Se ha creado una guía para estudiantes para ayudar en la instalación, el uso y la optimización.

El proyecto se divide en varias secciones. El Capítulo 2 describe la metodología, que incluye la recopilación de requisitos, la implementación, las pruebas y la evaluación por parte del usuario. El Capítulo 3 detalla el desarrollo de la aplicación de control de circuito de detección de temperatura, cubriendo sus características, arquitectura e interacción con

el usuario. El Capítulo 4 aborda aspectos prácticos, como la configuración del circuito, la calibración y el análisis de resultados. En el capítulo de conclusión, el proyecto resume los objetivos y hallazgos clave, destacando la usabilidad de la aplicación y su potencial para mejorar las experiencias de aprendizaje. Se reconoce el proceso de aprendizaje en la programación de Python y el diseño de GUI, enfatizando la importancia de la adaptabilidad.

En resumen, el proyecto ha logrado sus objetivos de trabajo principales al proporcionar una aplicación educativa (PyroDAQ) para la medida de la temperatura y la adquisición de datos. Ofrece una base para una mayor exploración y personalización en el campo, destacando el valor de las herramientas de código abierto y las experiencias prácticas en la instrumentación electrónica.

1. INTRODUCTION AND BACKGROUND

The motivation for this project stems from the significance of temperature sensing and data acquisition in various fields, particularly in electronics and the broader engineering domain. Temperature sensing is a fundamental aspect of modern engineering, impacting a wide range of applications from climate control to industrial processes, healthcare, and scientific research.

Accurate temperature measurements are crucial for ensuring safety, optimizing performance, and maintaining quality in various systems. To achieve this, temperature sensing circuitry plays a pivotal role. Examples of the importance of temperature sensing include the pharmaceutical industry, where accurate temperature measurement ensures the safe storage and transportation of pharmaceutical products to prevent degradation. In scientific research, precise temperature control is critical for experiments in biology and chemistry. In the automotive industry, temperature plays a key role in ensuring optimal engine performance and fuel efficiency. These examples highlight the diverse applications where temperature measurement and data acquisition are fundamental.

In the world of engineering, the use of National Instruments (NI) and Data Acquisition (DAQ) devices is notable. These devices enable interaction with a wide range of sensors and instruments, making them pivotal in experimental setups, automation systems, and quality control processes.

However, in classrooms, the accessibility and user-friendliness of these devices have often been overshadowed by complex and proprietary software, such as LabVIEW (Laboratory Virtual Instrument Engineering Workbench) [1], which conceals the intricate connection between hardware and software.

This project's motivation lies in addressing these challenges by leveraging the versatility and power of Python, an open-source programming language renowned for its simplicity and ease of use. Python's plethora of libraries makes it an ideal candidate for scientific

and engineering applications, including the potential to interface with NI DAQ devices, thereby revealing the bridge between hardware and software.

The culmination of this motivation is the development of a GUI-based Python application. This application aims to provide a user-friendly interface while being closely integrated with NI DAQ systems. By doing so, it seeks to empower users in the field of temperature sensing and data acquisition, making these processes more accessible, comprehensible, and efficient. This project represents a significant step towards enhancing the usability and accessibility of essential tools for engineers, researchers, and students in the realm of electronics and engineering.

1.1. A BRIEF INTRODUCTION TO THE NATIONAL INSTRUMENTS DAQ AND LABVIEW

National Instruments has been a prominent player in the engineering world for more than 40 years [2], providing tools for data acquisition and control systems. The key component for this research is the NI Data Acquisition (DAQ) hardware which has traditionally been used with the software LabVIEW.

LabVIEW is a widely recognized system design and development software platform created by National Instruments. It is commonly used in engineering and scientific fields for designing, testing, and implementing systems that involve measurement and control. It is renowned for its graphical programming approach, where users can create applications by connecting visual icons and wires, making it accessible to both engineers and scientists who may not have extensive programming backgrounds. It enables the development of custom graphical user interfaces (GUIs) and the creation of complex measurement and control routines for a wide range of applications.

NI DAQ systems are hardware devices designed to be the bridge between the real world and a computer. They facilitate the measurement and control of physical parameters such as current, temperature, pressure, or sound. At the heart of a DAQ system, there is signal conversion, input/output channels, signal conditioning, connectivity, and programmability.

Traditionally, LabVIEW has been the software of choice for interfacing with NI DAQ hardware, offering a way to design data acquisition and control applications. Engineers

and scientists could create custom GUIs and develop complex measurement and control routines without delving deep into low-level programming languages.

However, while LabVIEW has its merits, it also presents some challenges. Its learning curve can be steep for newcomers, and the visual programming paradigm may limit the accessibility of the underlying code for students and those seeking a deeper understanding of the hardware-software interaction. Moreover, LabVIEW is proprietary software, which can pose constraints on its distribution and customization.

This project aimed to address these challenges by developing an alternative solution.

1.2. PYTHON, FROM A VERSATILE PROGRAMMING LANGUAGE TO USES IN TEMPERATURE SENSING AND DATA ACQUISITION

Python, well-known for its versatility and approachability, stands apart in the realm of programming languages. While other languages such as C/C++ have better performance due to their lower-level nature and direct memory control [3], they require more intricate coding and a less comprehensive understanding upon initial inspection.

That’s why, in this context, Python's suitability for small-scale projects like the present one becomes clear, as it enables a direct approach without dealing with unnecessary complexities.

A distinctive facet of Python's utility emerges in its rich library ecosystem. It offers excellent integration for calculations through libraries like NumPy and SciPy, enabling swift development and easy implementation of mathematical operations. Along with others like matplotlib that permit visualizations and plots, parallel the functionalities offered by programs like MATLAB [4]. In the realm of electronic instrumentation, these libraries are particularly valuable. Here, precision mathematical calculations underpin a multitude of tasks, from signal analysis to data interpretation.

This relevance isn't just confined to calculations; it extends to the realm of hardware integration. This is most evident in fields where electronics and software converge, as exemplified by the integration of Python libraries tailored for controlling NI DAQ which this project is based on [5].

1.3. OVERVIEW OF TEMPERATURE SENSING CIRCUITRY

In the realm of temperature sensing, various circuit configurations are designed to cater to different levels of complexity and precision. These circuits serve as fundamental components in data acquisition and instrumentation.

Temperature sensing circuits are instrumental in enabling precise temperature measurements, a critical parameter in numerous applications spanning diverse industries. The choice of circuitry hinges on several key factors, including the required accuracy, the range of temperatures to be measured, and the specific application's demands.

At the foundational level, circuits can effectively employ sensors with a linear voltage output directly proportional to temperature in degrees Celsius. Their inherent simplicity makes them an excellent starting point for introductory temperature sensing experiments, providing a solid foundation in the field.

As one progresses into intermediate-level temperature sensing circuits, more advanced sensors such as thermistors or integrated digital temperature sensors come into play. These sensors offer enhanced accuracy but may require additional signal conditioning and calibration processes to ensure optimal performance and reliability.

In the course of this project, a deeper exploration of temperature-sensing circuits will be undertaken. Their principles, calibration methods, and practical applications will be thoroughly examined. Through this exploration, a comprehensive understanding of the temperature measurement techniques that form the foundation of data acquisition and instrumentation in diverse fields will be achieved.

1.4. IDENTIFICATION OF GAPS IN THE TRADITIONAL APPROACHES THAT THE CURRENT RESEARCH AIMS TO ADDRESS

As previously mentioned, a significant gap exists between tools like LabVIEW and their users, creating a noticeable disconnect between software and hardware components. This issue is particularly prevalent among students who are venturing into practical applications for the first time, transitioning from the realm of theoretical knowledge to hands-on experience. This transitional phase in their education demands a clear understanding of the intricate connections between the hardware they're working with,

the circuits they're studying, and the software that generates the results. Unfortunately, existing tools like LabVIEW often fall short of providing this level of transparency.

For students, constructing interfaces within LabVIEW for their experiments has been the norm. While this highlights the software's versatility, it imposes a dual challenge on students. Not only must they bridge the gap between theory and practice in their projects, but they are also compelled to construct the very interfaces they rely on. Regrettably, this construction phase usually leaves them with limited insights into the underlying mechanisms behind the GUI.

1.5. OBJECTIVES

The primary aim of this project is to streamline this construction process by offering a straightforward application. This application not only simplifies the user experience but also allows users to delve into its inner workings, providing a unique opportunity to explore the connections between hardware and software, thereby addressing the aforementioned gaps in traditional approaches.

Furthermore, another additional objective is to emphasize key concepts in the broader field of electronic instrumentation, with a particular focus on temperature measurement. These concepts can be experimentally explored using the developed application, allowing for a deeper understanding of topics such as calibration, sensitivity, resolution, load regulation, self-heating, and more.

Both objectives inherently entail the creation of a student guide to assist them in the installation, usage, and optimization of the provided application.

1.6. OVERVIEW OF THE STRUCTURE OF THE THESIS

The project is divided into several key sections that collectively form a comprehensive exploration of the research, development, and application of this educational tool.

In chapter 2, the methodology employed for the research and development of the educational application is outlined. It begins with an overview of the methodology, followed by a discussion of requirement gathering and analysis. The chapter then delves into the implementation process, covering the selection and description of the NI DAQ system and temperature sensor used, as well as an overview of the programming

environment, Python, and the software components utilized. The methodology also includes a comprehensive strategy for testing and quality assurance, as well as a section on user evaluation and feedback.

Chapter 3 constitutes the heart of the thesis, detailing the development of the temperature sensing circuit control application. It begins with an introduction to the application's purpose and significance. The subsequent sections provide an in-depth exploration of the application's features and capabilities, the design of the user interface, the high-level architecture, and the GUI development process. It also covers the management of user interaction and control logic, including the integration of the NI DAQ and temperature sensing functionality. The chapter concludes by demonstrating the user interaction flow, error handling strategies, testing, debugging, and validation processes, and offers suggestions for potential future research and development directions.

Chapter 4 is dedicated to the practical aspects of the research. It begins with a detailed description of the circuit setup used for temperature sensing and data acquisition. Subsequently, it outlines the calibration and validation procedures for the temperature sensor. The chapter culminates with the presentation of the results and analysis, which include considerations regarding the program's ability to enhance student engagement in experimental setups.

In the concluding chapter, the thesis provides a summary of the research objectives and key findings, emphasizing the successful development of the educational application and its usability for students. It also explores the program's potential to enhance learning experiences in temperature sensing and data acquisition, shedding light on the broader implications of this innovative approach to education. The chapter closes with concluding remarks that highlight the project's achievements and its potential for further expansion and exploration.

Additionally, the thesis includes three appendices: Appendix A contains image attributions used in the application, Appendix B features the comprehensive student's guide, and Appendix C provides access to the code repository for this project.

2. METHODOLOGY

2.1. OVERVIEW OF THE METHODOLOGY

The methodology employed in this project played a pivotal role in addressing a significant challenge that loomed at the beginning of the project: the mastery of Python programming and the intricacies of constructing an application with a GUI, all within the context of a limited background in the subject matter. This endeavor marked a journey of trial and error, a path defined by the necessary adaptation to the new concepts encountered at every step of the development process.

Central to the project's achievement was the necessity to acquire proficiency in the Python programming language from the ground up. Despite the absence of prior Python familiarity, the project was facilitated by the foundation of advanced competence in both C and C++. This advanced foundation played a crucial role in guiding the learning process, therefore, through a combination of self-guided exploration, interactive coding exercises, and reference to the established online resources in the vibrant Python community, a foundational competence in Python was cultivated.

The considerable assistance rendered by *Python for Science and Engineering* by Hans-Petter Halvorsen is properly noted [4]. This resource played a significant role in providing a structured pathway for acquiring Python skills in the context of mathematical applications, data acquisition, and the control of National Instrument devices. The clear explanations, practical exercises, and readily available resources contributed significantly to the foundation of this project. This learning phase created the groundwork for the subsequent stages, setting the stage for the development of the application.

The challenge, however, transcended the realm of programming proficiency. The construction of an application with a GUI posed a different set of challenges, particularly given the relatively modest exposure to GUI development. Consequently, selecting a suitable GUI framework became crucial for addressing this issue. Therefore, the decision was made to opt for PySimpleGUI. As its name suggests, PySimpleGUI is a straightforward and accessible wrapper, with many online resources for learning and developing [6]. Sections to come will delve into a comprehensive exploration of the advantages and disadvantages presented by this wrapper.

This learning process necessitated a strategic shift towards a trial-and-error approach; thus, the construction of the application's functionality and user interface became a dynamic process of experimentation and adaptation. With each iteration, valuable insights were gleaned, serving as steppingstones toward a refined and user-friendly GUI.

Throughout the process, it became evident that adhering to a key principle of adaptation was crucial. Every obstacle encountered provided a learning opportunity, and each advancement was marked by the assimilation of new skills and insights. It consequently became apparent that, for effective programming of any user interface, the formulation of a well-structured plan with rough sketches detailing the arrangement of elements within each window and their interactions was of paramount importance to ensure efficiency coding-wise.

This process of perpetual learning was not confined to the realms of programming and GUI design alone; it encompassed a broader spectrum of skills, from debugging intricate code segments to integrating user feedback into the application's evolution.

In summary, the methodology embraced in this project was an example of adaptability and development. It provided a platform for tackling the complexities of Python programming and GUI design.

2.2. REQUIREMENT GATHERING AND ANALYSIS

The process of requirement gathering and analysis for the application was rooted in the subject laboratory session for which the application is intended to be used. The foundational essence of this lab session served as the basis for formulating the application's requirements. The main goal was to transition the functionalities initially executed in LabVIEW to a dedicated application environment.

The requisites encompassed the ability to effectively recognize and manage the NI DAQ, facilitate the computation of calibrations for the designated temperature sensor circuit, provide the means for utilizing these calibrations for data acquisition purposes, and present a user-friendly interface for comprehending and controlling the acquired data and the associated DAQ functions.

Furthermore, the process included catering to the specific needs of the students. This was achieved by engineering an application that remains focused on its primary purpose. By

simplifying its features, the application facilitates a practical exploration for students to apply their theoretical understanding of temperature sensor circuits. This approach mitigates the requirement for them to divide their focus between mastering diverse software tools, such as LabVIEW, and comprehending the core subject matter.

2.3. IMPLEMENTATION

An overview of the implementation process encompassing both hardware and software components is presented in this section. The integration of NI DAQ, temperature sensors LM35 and PT100, alongside the utilization of Python and PySimpleGUI, constitutes the focal points of this phase. The subsequent sections will provide an in-depth exploration of each component, explaining their roles and collaborative interactions within the application's architecture.

2.3.1. DESCRIPTION OF THE NATIONAL INSTRUMENT DAQ SYSTEM AND TEMPERATURE SENSOR USED

The ensuing section offers a comprehensive portrayal of the specific hardware components central to the study's implementation, focusing on the NI DAQ system and the temperature sensors employed. The DAQ system of choice was the entry-level, plug-and-play USB series. During the project's construction, the primary DAQ system utilized was the USB-6001, notable for its robust capabilities. The USB-6001 is often considered a cost-effective solution for basic data acquisition needs. The USB-6001, characterized by its compact form factor, facilitated seamless integration with the Python-based application. It offers 8 single-ended (4 differential) analog input channels and includes 2 analog output channels for signal generation. Specifications include its 14-bit analog-to-digital converter (ADC), digital input/output capabilities, and a moderate sampling rate (20 kS/s maximum) [7] and [8]

Furthermore,
compatibility with various DAQ systems was included in the implementation scope, such as the USB-6002 and USB-6211. These DAQ systems, while sharing commonalities with the USB-6001, offer distinctions in terms of analog input channels, sampling rates, and voltage ranges, catering to diverse experimental demands.

Regarding temperature sensors, the preliminary phase incorporated the LM35, a simple yet effective sensor suitable for initial testing scenarios. The LM35 facilitated straightforward temperature readings in basic circuit configurations. However, as the project's complexity grew, a shift toward a more advanced sensor took place. The Pt100, a platinum resistance temperature detector (RTD), emerged as the sensor of choice for enhanced precision and accuracy.

The LM35 and Pt100 sensors present notable differences. The LM35 operates linearly within a specific temperature range and offers direct temperature-to-voltage conversion, making it suitable for rudimentary applications. Contrariwise, the Pt100 exhibits a wider temperature range, increased accuracy, and a resistance-based response that necessitates specialized circuitry, such as a Wheatstone bridge, to convert resistance changes into voltage values. The way this bridge is configured in terms of power supply and the values of fixed resistors will ultimately determine its sensitivity to changes in temperature.

One could emulate the behavior of a Pt100 at a specific temperature by simply replacing it in the circuit with a precision resistor of the same value as the Pt100 at that temperature. This opens the possibility of emulating the circuit's behavior for various Pt100 temperatures, a crucial aspect for implementing and evaluating calibration methods, thereby aptly accommodating the application's evolution.

2.3.2. OVERVIEW OF THE PROGRAMMING ENVIRONMENT (PYTHON) AND SOFTWARE COMPONENTS UTILIZED

The subsequent segment offers an encompassing insight into the programming environment leveraged for the project, primarily focusing on Python and the software components integral to its functionality.

Central to the project's interface design was the adoption of PySimpleGUI, a GUI wrapper acclaimed for its intuitive nature and robust capabilities. PySimpleGUI encapsulates the complexity of GUI development, serving as a bridge between the programmer and the interface.

Further enhancing the programmer experience, PySimpleGUI effectively wraps Tkinter, an established GUI toolkit [9]. By doing so, PySimpleGUI harnesses the strengths of Tkinter's vast functionality while simplifying its implementation thus making the learning curve not as steep. This choice was motivated by PySimpleGUI's proficiency in striking

a balance between accessibility and power, making it an ideal fit for this project's objectives.

Furthermore, an indispensable driver, NI-DAQmx, is central to this project for its role in enabling interaction with NI DAQ devices [10]. As a product of National Instruments, NI-DAQmx presents a Python interface for seamless communication with DAQ hardware, facilitating data acquisition, control operations, and integration with other Python libraries.

Arithmetic and data manipulation was facilitated by the incorporation of libraries like NumPy, SciPy, and Matplotlib. NumPy [11] and SciPy [12] were employed to perform mathematical computations, specifically in deriving equations for linear and non-linear equations to interpret the data. These libraries enabled the formulation of mathematical models that accurately described the relationships present within the dataset. Additionally, Matplotlib [13] enabled visually appealing data visualization. These libraries collectively consolidated the project's capacity to manage and represent data effectively.

In summary, the programming environment was meticulously curated, with PySimpleGUI championing intuitive yet robust interface design, NI-DAQmx orchestrating seamless DAQ device communication, and the arithmetic libraries fostering data manipulation and visualization.

2.4. TESTING AND QUALITY ASSURANCE STRATEGY

This section explains the planned testing approach undertaken to establish the application's functionality, reliability, and performance. It involves various tiers of testing, encompassing unit testing, integration testing, and system testing, with each stage designed to guarantee careful evaluation.

Unit testing was used to examine individual components in isolation and was typically performed as the code was developed and each portion was completed.
Specific functions and methods of the code were tested to ensure that in isolation they worked as intended and produced accurate results. This ensured that when having to work together with other units, a solid foundation was established. These units of the

application, such as temperature data acquisition, GUI interactions, and calibration calculations

Integration testing involved verifying the interplay between the application's components to ensure that the individual units of code worked together cohesively as a larger integrated system. Interaction scenarios, encompass user input through the GUI, data acquisition via the DAQ system, and the subsequent processing and display of results. Through this process, seamless cooperation can be ensured between diverse modules.

System testing, an evaluative approach that scrutinizes the application in its entirety, was executed through a series of thoughtfully designed scenarios. These scenarios were crafted to emulate real-world user interactions and edge cases to assess the application's coherence and resilience.

These scenarios encompass the emulation of normal user behavior, extreme temperature values, rapid user interactions, erroneous input, compatibility tests conducted with different DAQ systems, and the absence of such.

Collectively, these scenarios underscored the application's ability to manage diverse user interactions, ensuring a comprehensive evaluation of its functionality and reliability.

The process of bug identification, tracking, and resolution followed a structured approach. Detected issues were logged and assigned priority levels. The tracking system ensured that each issue was traced from identification to resolution. This methodology ensured that bugs were systematically addressed, guaranteeing the application's robustness.

In essence, the orchestrated testing approach encompassed a spectrum of meticulous examination, extending from granular unit tests to comprehensive system validation. This method proved invaluable as it enabled the achievement of a polished and dependable outcome. By swiftly pinpointing issues without the inefficiencies of troubleshooting, the established timelines were adeptly adhered ensuring successful project completion.

2.5. USER EVALUATION AND FEEDBACK

The process of user evaluation and feedback collection was used in assessing the application's usability and effectiveness. This phase involved the simulation of diverse user scenarios, effectively mirroring real-world usage scenarios to ensure a comprehensive assessment.

Following the guidelines outlined in the provided student guide, a designated group of users interacted with the application, exploring its features as intended. This emulation provided valuable insights into the application's user-friendliness, navigational logic, and overall effectiveness in achieving its intended purpose.

Contrariwise, to gauge the application's robustness and error-handling capabilities, users were given a degree of freedom to interact with the application as they saw fit. This exploratory approach aimed to identify any potential vulnerabilities, unanticipated usage patterns, or points of failure. By observing how the application performed in such scenarios, the evaluation encompassed a comprehensive examination of its resilience and error-catching mechanisms.

Feedback like issues and suggestions was incorporated in the same mannerism that bugs were tackled in the testing stage. By logging and prioritizing, solutions were devised, tested, and integrated into the application. This feedback was subsequently used as a foundation for enhancing the application and, in doing so, aligning it more closely with users' expectations and workflow.

Furthermore, the project was also uploaded to GitHub, providing a platform for potential future feedback and contributions from the open-source community, which may further improve the application's functionality and usability. To this day, feedback hasn't been received, but suggestions are awaited, and they will be carefully considered for the project's ongoing development and refinement.

3. DEVELOPING THE APPLICATION

This section introduces the temperature sensing circuit control application, designed to establish an interface between a NI DAQ device and a temperature sensor using Python. This integration facilitates the calibration of the circuit and acquisition of accurate temperature readings, offering a solution for temperature sensing circuit control.

3.1. INTRODUCTION TO THE TEMPERATURE SENSING CIRCUIT CONTROL APPLICATION

The application's architecture is made to bridge the gap between hardware and software, enabling students to access temperature data easily with precision. Its role in this context is key, as its simplicity enhances the efficiency and accuracy of temperature measurement processes without the user having to do anything else.

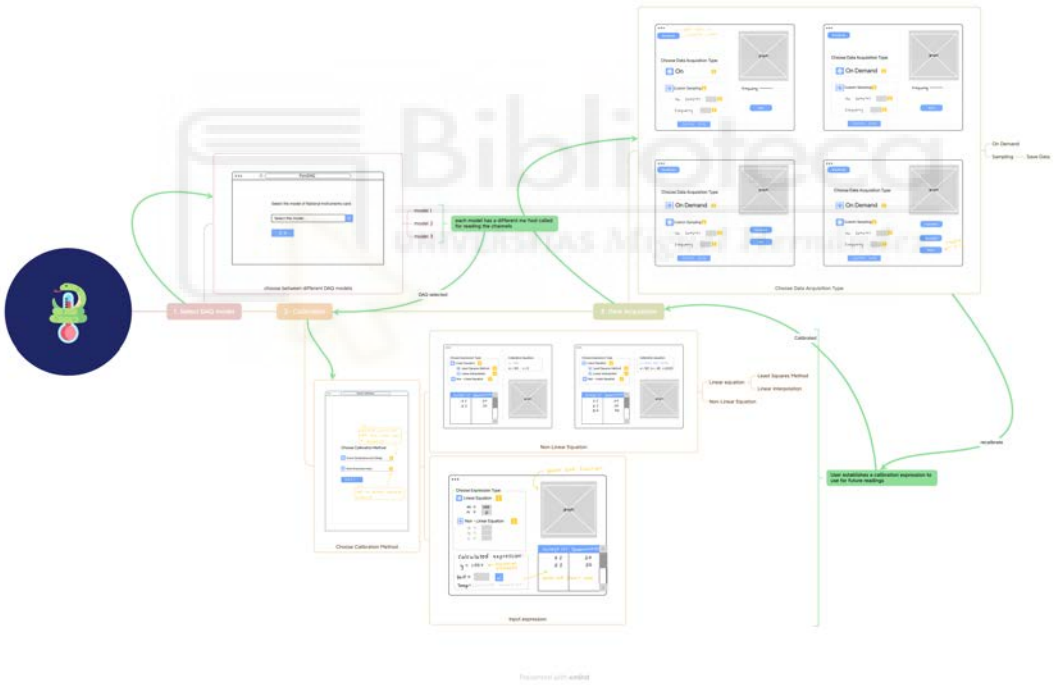

Fig. 3.1.1 Initial wireframe of PyroDAQ application design

A prominent feature of the application is calibration, a key mechanism that serves as the foundation for temperature measurement precision. Through the application's intuitive interface, students can establish a calibration line or curve, correlating known temperature inputs with the sensor's corresponding readings. They can also choose to establish an already known calibration. This calibration serves as the foundation for converting raw

sensor data into reliable temperature values, ensuring the accuracy of acquired measurements.

Additionally, the application offers data acquisition capabilities, interfacing with the NI DAQ device. This integration empowers students to monitor temperature fluctuations in real-time, enabling dynamic analysis and experimentation within temperature-sensitive circuit control environments.

The significance of the developed application is rooted in its capacity to simplify the intricacies associated with building the interface for temperature sensing circuit control from scratch. By facilitating data acquisition and interpretation, it emerges as a valuable tool for students as it bypasses any additional designs in LabVIEW and directly offers the necessary tools.

Its pragmatic utility extends beyond temperature measurement, encompassing a comprehensive educational tool that bridges theory and practical implementation. This powerful tool not only allows for the observation of Python in action within an electronics environment but also serves as a dynamic platform for gaining hands-on experience and insights into real-world applications.

3.2. APPLICATION FEATURES AND CAPABILITIES

The emphasis revolves around facilitating precision in temperature sensor calibration and enabling real-time data acquisition. Together, these essential features improve the application's ability to handle the complexities of temperature sensing in circuit control situations.

One significant functionality center on establishing a correlation between raw sensor readings and known temperature values. This process, integral to the application, results in the calibration of the temperature sensor (Fig. 3.2.1). The significance of this calibration becomes pronounced in real-world scenarios, where precise temperature monitoring is imperative for effective circuit control.

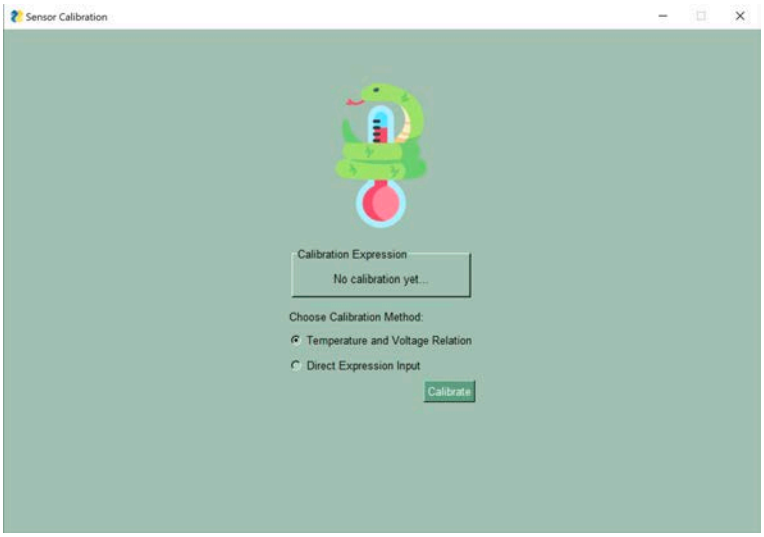

Fig. 3.2.1 Calibration options

This calibration process is designed to be intuitive and flexible. Users are granted the capability to input calibration coefficients directly using a known mathematical expression (Fig. 3.2.2). This empowers users with the freedom to tailor the calibration to their specific needs. Moreover, the application provides a practical feature that enables users to test the accuracy of the entered calibration while still in the input window, ensuring confidence in the calibration's reliability (Fig. 3.2.3).

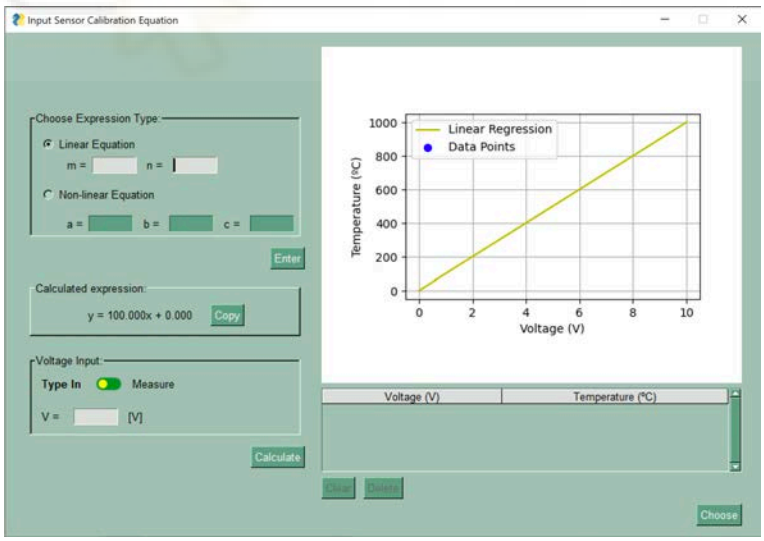

Fig. 3.2.2 Direct calibration input for a LM35 temperature sensor example.

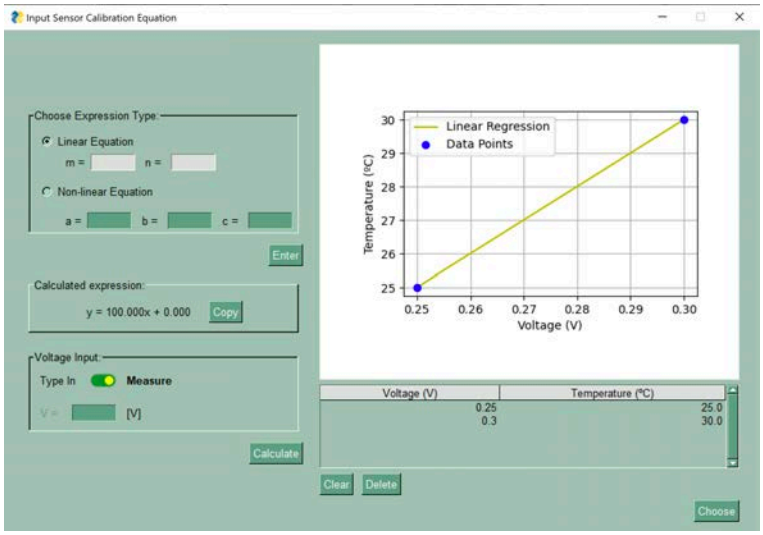

Fig. 3.2.3 Testing the calibration.

Alternatively, the application offers an avenue for calibration selection by providing a known temperature and voltage correlation (Fig. 3.2.4). Users can choose the calibration that best fits their understanding and requirements. This method aligns with empirical learning, allowing students to witness the practical implications of their calibration choices.

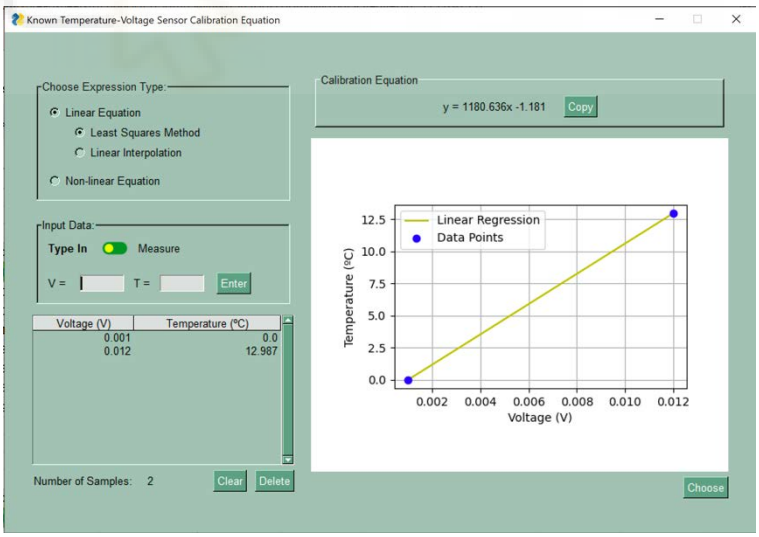

Fig. 3.2.4 Known temperature-voltage correlation calibration

Calibrations within the application cater to diverse scenarios and needs, accommodating both linear and non-linear calibrations (Fig. 3.2.7). For linear calibrations, users are equipped with the option to choose between two calculation methods: the least squares method (Fig. 3.2.5) and interpolation between user-selected data points (Fig. 3.2.6). This

versatility in calibration methodologies ensures adaptability to varying user preferences and complexities.

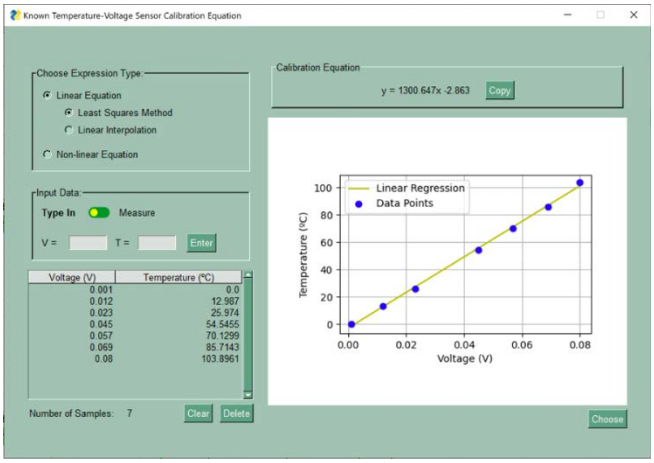

Fig. 3.2.5 Linear Calibration, Least Squares Method

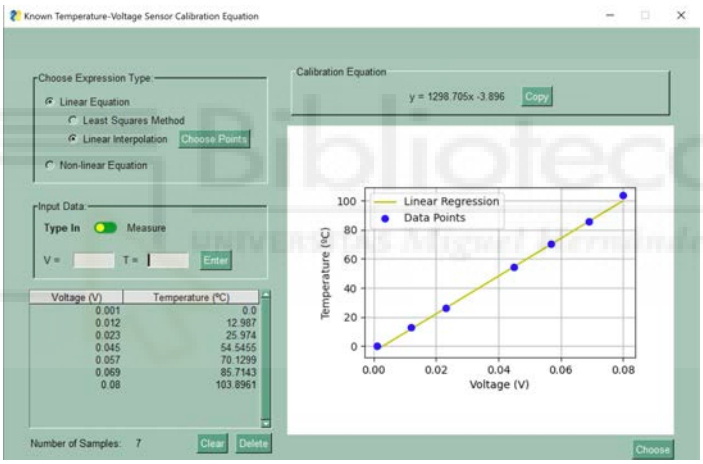

Fig. 3.2.6 Linear Calibration, Linear Interpolation

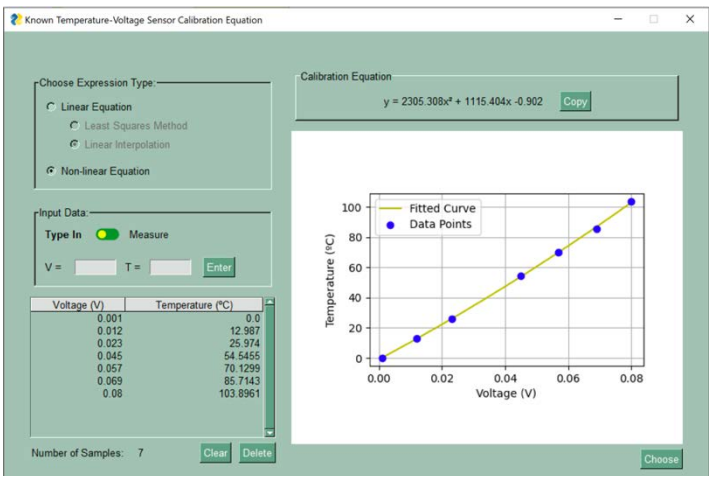

Fig. 3.2.7 Non-Linear Calibration

Every calibration set by the user is automatically saved and stored in logs. These saved calibrations can be readily accessed for future trials and experiments (Fig. 3.2.8), eliminating the need for repeated calibration setups.

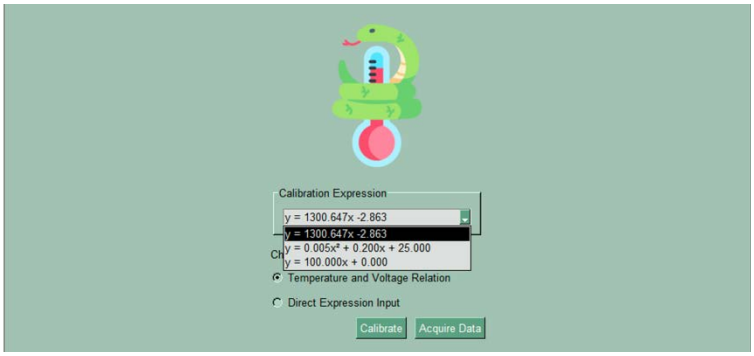

Fig. 3.2.8 Calibration log

In essence, the application's calibration functionality enhances precision, fosters active learning, and provides an array of options to suit different scenarios, ultimately enriching the educational and practical value of the platform.

Furthermore, the application integrates the calibration process with real-time temperature data acquisition. By interfacing with the NI DAQ device, users are bestowed with the ability to procure temperature data within dynamic environments. This pairing of data acquisition is accompanied by a simultaneous process of data logging (Fig. 3.2.9) and real-time plotting (Fig. 3.2.10), thus elevating the application's utility.

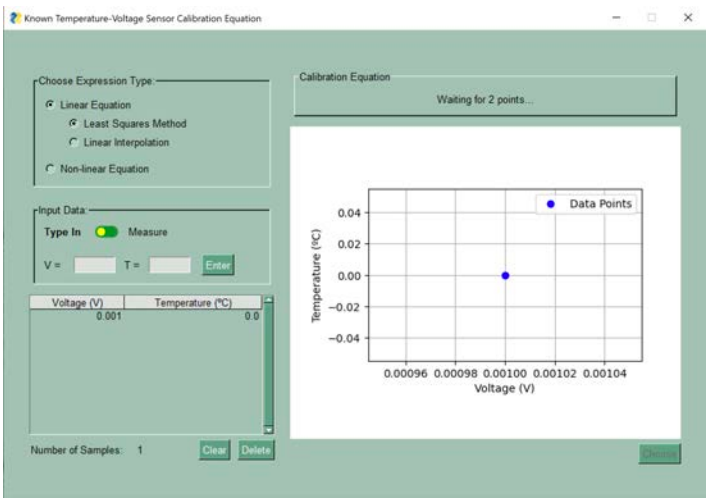

Fig. 3.2.9 Example of data logging in known temperature and voltage correlation

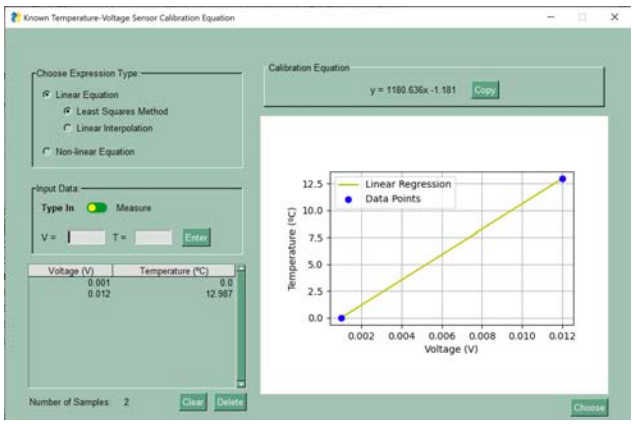

Fig. 3.2.10 Further Examples of Data Logging, now with a Plotted Line

This multifaceted functionality becomes particularly noteworthy in scenarios where the monitoring of temperature variations is in real time is. This empowerment extends to both educational and practical realms, enabling students and users to closely track temperature fluctuations as they occur.

Diving deeper into data acquisition, the application offers users the flexibility to choose between two distinct modes. The first mode, "On Demand" allows users to trigger data acquisition at their discretion. This mode is characterized by indefinite data acquisition, enabling users to gather data continuously until they choose to halt the process. During this ongoing data acquisition, users retain the freedom to dynamically adjust the time interval between successive acquisitions (Fig. 3.2.11) and (Fig. 3.2.12). This adaptability is mirrored in real-time data plots, enabling users to witness immediate reflections of their chosen time intervals on the evolving data plot.

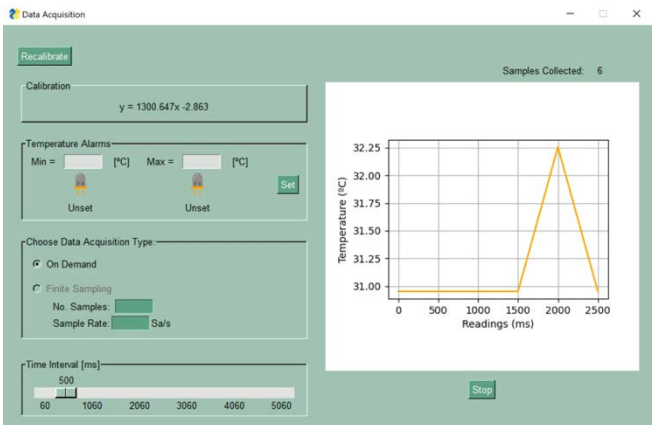

Fig. 3.2.11 On Demand Option with 500ms Time Interval, Data Acquisition

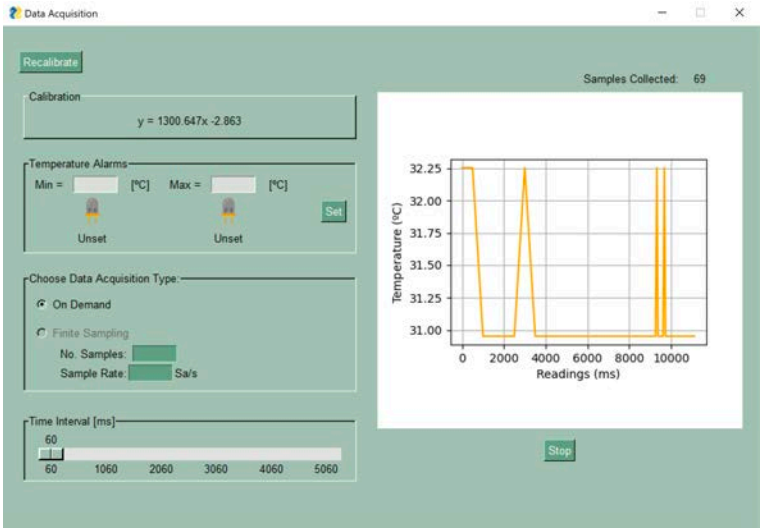

Fig. 3.2.12 On Demand Option with 60ms Time Interval, Data Acquisition

Alternatively, the application caters to scenarios that demand a finite sampling approach. In this mode, termed "Finite Sampling" users define both the total number of samples they wish to acquire and the desired sample rate (Fig. 3.2.13). This approach offers control over the data collection process, allowing users to gather a predetermined amount of data with temporal intervals.

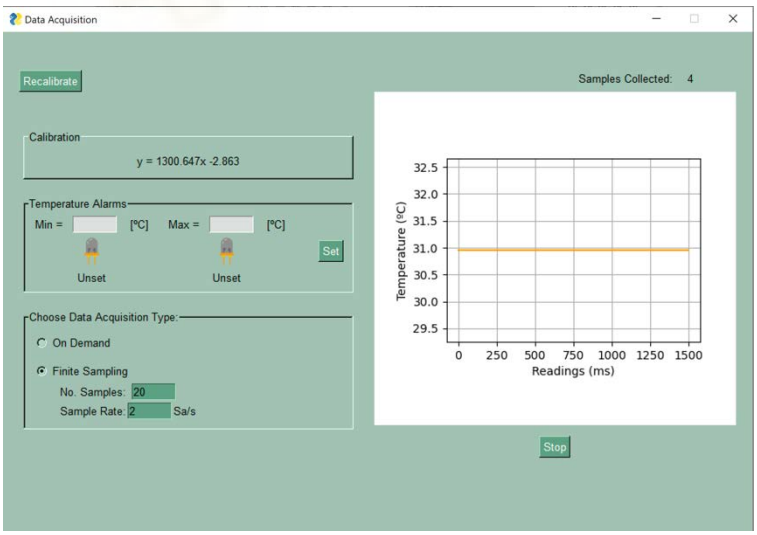

Fig. 3.2.13 Finite Sampling Example with 20 samples and 2 Sa/s, Data Acquisition

Moreover, the application's monitoring functionalities extend to establishing alarm limits for both lower and/or upper-temperature thresholds. These alarms trigger as soon as

temperature readings breach the set limits (Fig. 3.2.13). In this manner, users are promptly alerted to deviations from the desired temperature range.

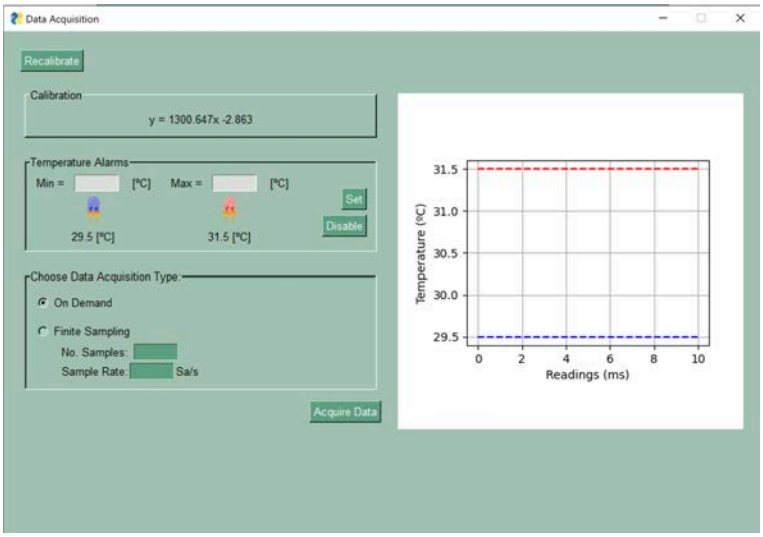

Fig. 3.2.14 Example of Alarms Set at 30.5ºC and 31.5ºC and Maximum Alarm Going Off, Finite Sampling, Data Acquisition

All collected data, along with the triggered alarms and corresponding readings, can be saved, offering a comprehensive record of the system's behavior (Fig. 3.2.15) and (Snippet 3.2.1). This comprehensive suite of monitoring features not only ensures real-time data collection but also equips users with valuable insights by logging alarm-triggered events and temperature parameters.

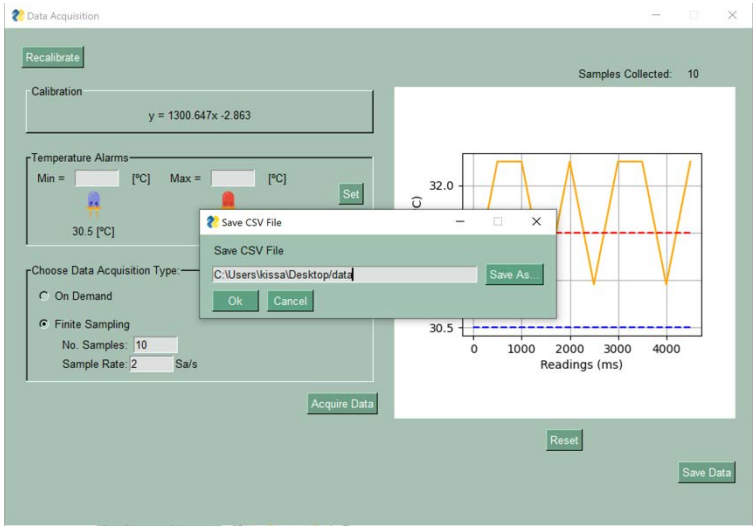

Fig. 3.2.15 Saving Data as a CSV File

```
13/09/2023 13:40:00.582277

CALIBRATION
y = 1300.647x -2.863

PARAMETERS
Number of samples,Sample rate [Sa/s]
10,2.0

ALARM LOGS
Min alarm,Max alarm
30.5,31.5
Alarm Type,Temperature,Time Interval
Above Maximum,32.254,500
Above Maximum,32.254,1000
Above Maximum,32.254,2000
Above Maximum,32.254,3000
Above Maximum,32.254,3500
Above Maximum,32.254,4500

DATA
Voltage [V],Temperature [∫C]
0.026,30.954
0.027,32.254
0.027,32.254
0.026,30.954
0.027,32.254
0.026,30.954
0.027,32.254
0.027,32.254
0.026,30.954
0.027,32.254
```

Snippet 3.2.1 CSV File 'data.csv'

3.3. DESIGNING THE USER INTERFACE

3.3.1. ICON DESIGN AND GUI AESTHETICS

The icon design and user interface aesthetics were considered during the development of the application. Several design choices were made to enhance user-friendliness and appeal, particularly with a focus on student and youth engagement.

The primary color chosen for the application was green (Fig. 3.3.1). This choice was deliberate, as it aligns with the Python programming language, a key component of the application. The use of green was intended to make the application more approachable and relatable to students.

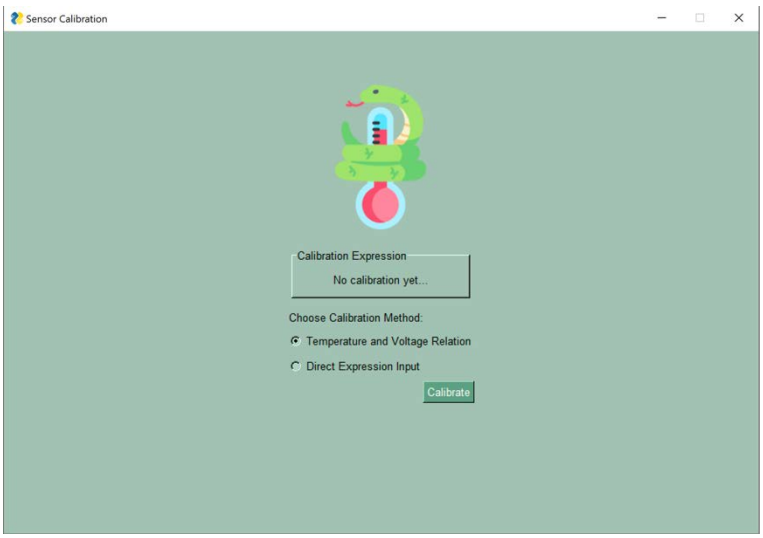

Fig. 3.3.1 PyroDAQ icon in place with the green layout

The application's logo is a central element of its visual identity. The logo features a snake wrapped around a thermometer (Fig. 3.3.2). This design was selected to symbolize the application's Python-based functionality for temperature sensor circuit control. The snake represents Python programming, while the thermometer signifies temperature sensing—a direct reflection of the application's purpose. Notably, the logo was intentionally designed in an emoji-like style. This decision was made to make it more appealing and relatable to younger users, including students. The emoji-style design aims to alleviate the apprehension associated with the traditionally complex world of instrumentation towards a more familiar and friendly aesthetics of youth-oriented applications.

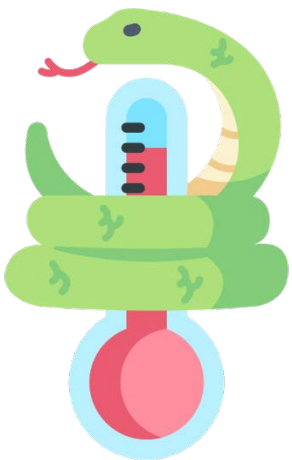

Fig. 3.3.2 PyroDAQ icon

The application's icon and other graphical assets were shaped by the author, from a selection of copyright-free images. These images were used as foundational elements for the design but were adapted to create cohesive visual elements. Proper attribution for the original images used in the icon and other assets is provided in the "APPENDIX A: ASSETS AND ATTRIBUTION" section of the appendix.

In keeping with the overall design philosophy, other graphical assets within the application were also fashioned in an emoji-like style. This consistent approach ensures that the visual elements maintain a cohesive and engaging appearance, enhancing the user experience.

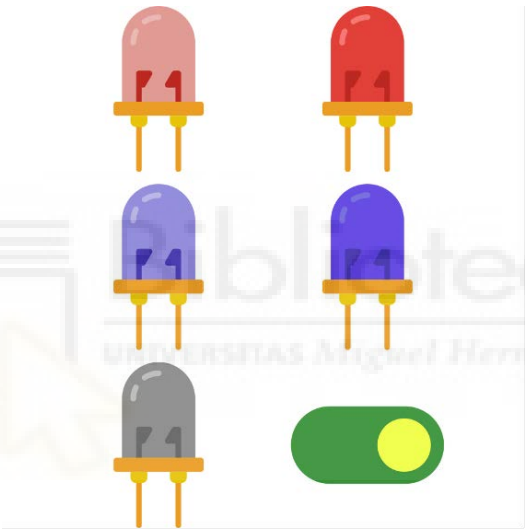

Fig. 3.3.3 PyroDAQ assets for temperature alarms and toggle

The combination of these design choices, from the color scheme to the logo and icon design, contributes to an interface that is both visually appealing and aligned with the educational goals of the application. It aims to create a welcoming and accessible environment for students and users of all backgrounds to explore the world of temperature sensing and data acquisition.

3.3.2. GUI LAYOUT AND COMPONENTS

Within the domain of designing the user interface, the application's core considerations revolve around optimizing user experience, ensuring accessibility, and seamlessly guiding users through the application's functionalities.

The development process prioritizes the ease of use for students, making accessibility and usability paramount. The application's design aims to minimize user effort by creating a seamless and intuitive user journey.

The necessary components and controls within the user interface have been outlined to facilitate effective visualization. The application embraces real-time visualization, showcasing crucial inputs such as calibration expressions, voltage readings, and calculated temperatures. This dynamic display provides users with instant feedback, enhancing their engagement and understanding.

A delicate balance between user freedom and guided functionality is maintained within the interface design. While users are granted the flexibility to navigate and utilize the application as intended, certain controls are strategically embedded to ensure a coherent experience. For example, buttons and inputs are thoughtfully hidden or revealed, disabled and abled based on the context, thereby preventing user confusion, and streamlining the process.

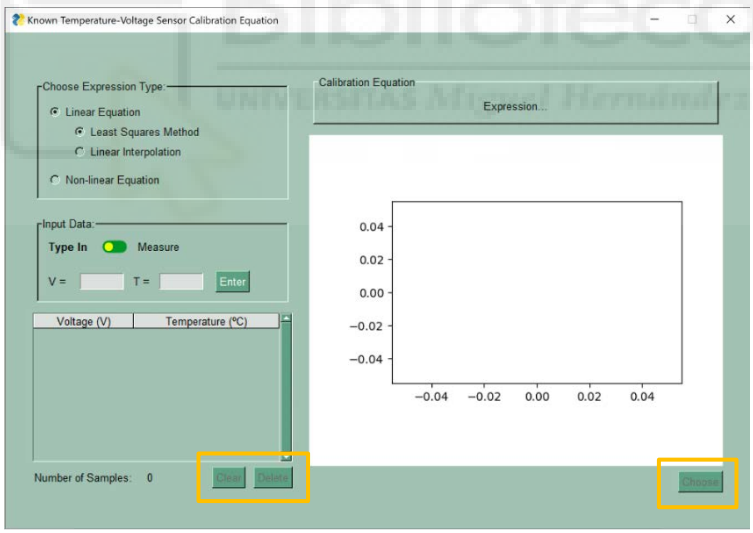

Fig. 3.3.4 Disabled Buttons When There is No Data or Calibration, Known Temperature-Voltage Correlation Calibration

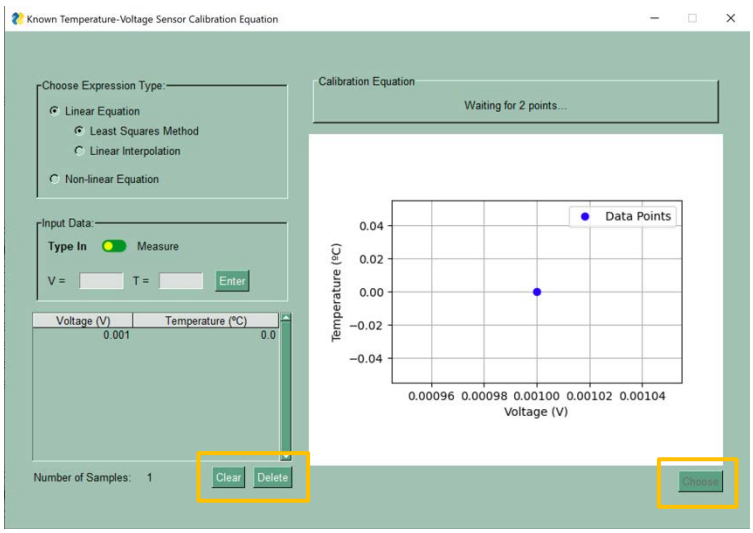

Fig. 3.3.5 Enabled button when there is data, disabled when there's no calibration, known temperature-voltage correlation calibration

This is observed in examples like Fig. 3.3.4 where if there's no data introduced, the "Clear" and "Delete" buttons are disabled, since there is no data, there's no need to edit the entries. Once data has been introduced, in Fig. 3.3.5 one can note that the "Clear" and "Delete" buttons are enabled. In both cases "Choose" remains disabled since this sets the calibration equation. Since there aren't enough points for the equation to be calculated, it doesn't make sense for the user to select it, so the option isn't available for the user. Thus, the user is unconsciously guided through the steps without requiring unnecessary interactions.

The intuitive nature of the interface is achieved by emulating a workflow that mirrors the traditional process. The application's flow guides users through familiar steps, ensuring a sense of continuity. This approach resonates particularly with users accustomed to following the structured workflow of manually calculating these scenarios, aligning the application's design with their expectations.

The process of designing and integrating these diverse components adheres to the natural progression of actions. The interface's layout follows a deliberate sequence, featuring components aligned from left to right and top to bottom. This organization reflects the logical progression of user inputs, enhancing clarity and coherence.

For instance, in cases of known voltage-temperature calibration, the interface design corresponds with the flow of decision-making. Users are prompted in the left column to choose the linearity of calibration, specify voltage input preferences, and log data points (Fig. 3.3.6). Concurrently, the interface updates the right column to reflect these inputs, displaying the expression being calculated and plotted alongside the logged data points (Fig. 3.3.7). This real-time feedback ensures users remain aware of their actions' outcomes.

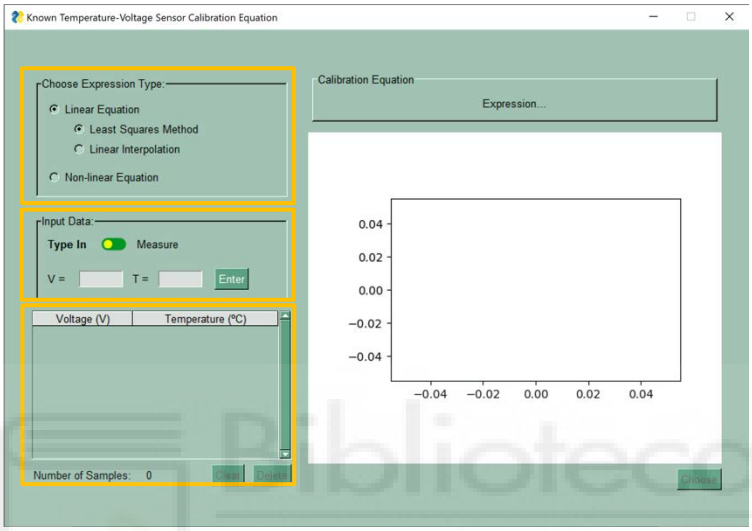

Fig. 3.3.6 Action Sequence for choosing linearity of calibration, data input, and data management in the left column, known temperature-voltage correlation calibration

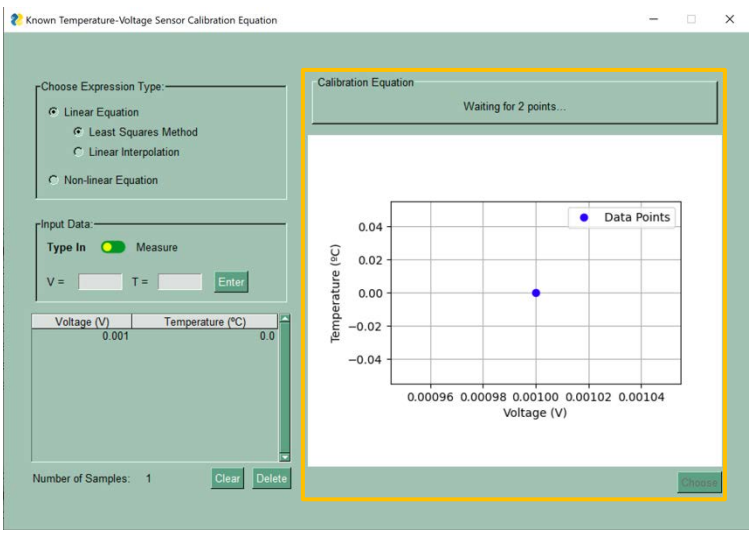

Fig. 3.3.7 Parallel updating of user interaction in the right column, known temperature-voltage correlation calibration

The user interface design unites accessibility, functionality, and familiarity. By maintaining an equilibrium between user autonomy and guided interaction, the application bridges the gap between user expectations and its functionalities. The interface's logical layout and real-time visualizations enhance user engagement and comprehension.

3.4. OVERVIEW OF THE HIGH-LEVEL ARCHITECTURE

The project's file hierarchy forms the backbone of its organization, fostering a clear distinction between logic and GUI components. This hierarchical structure ensures that the code is organized and modular, allowing different aspects of the application to work together while maintaining separation where necessary. The project's directory structure is as follows:

```
PyroDAQ/
├── assets/
│   ├── alarm_max_off.png
│   ├── alarm_max_on.png
│   ├── alarm_min_off.png
│   ├── alarm_min_on.png
│   ├── alarm_unset.png
│   ├── icon_big.png
│   ├── switch_off.png
│   └── switch_on.png
├── src/
│   ├── app/
│   │   ├── appCalibrationMethod.py
│   │   ├── appDAQ.py
│   │   ├── appDataAcquisition.py
│   │   ├── appExpressionInputCalibrate.py
│   │   └── appTempVoltCalibrate.py
│   ├── gui/
│   │   ├── guiCalibrationMethod.py
│   │   ├── guiDAQ.py
│   │   ├── guiDataAcquisition.py
│   │   ├── guiExpressionInputCalibrate.py
│   │   └── guiTempVoltCalibrate.py
│   ├── calibrationTools.py
│   ├── daqTools.py
│   └── guiTools.py
└── main.py
```

Snippet 3.4.1Project directory structure

The high-level architecture of the application can be understood through a hierarchical organization, divided into three distinct phases: the setup phase, calibration, and data

acquisition. This structure underscores the systematic flow that guides users through the application's functionalities.

At its core, the application's architecture is tripartite, rooted in the main script that serves as the epicenter of all operations. This main script initiates the setup phase, from which the subsequent branches emanate (Snippet 3.4.2). The branching nature of the architecture ensures a modular approach, enhancing both organization and scalability.

```
def main():

    # --- DAQ SELECTION ---
    niDAQ = daq.run_select_daq()
    while not niDAQ.is_exit_requested():
        # --- CALIBRATION ---
        calibration_method.run_calibrate(niDAQ)
        if niDAQ.is_exit_requested():
            niDAQ.exit()
            continue
        # --- ACQUIRE DATA ---
        data_acquisition.run_data_acquisition(niDAQ)

[...]
```

Snippet 3.4.2Branching in main function './main.py'

The architecture's design revolves around a symbiotic relationship between the logical components and the GUI. Each phase of the application bifurcates, with logical components branching towards various functions while remaining interlinked with the GUI (Snippet 3.4.3). This arrangement fosters a cohesive user experience where GUI elements and logical operations complement each other.

```
[...]

def run_select_daq():
    while True:
        try:
            # creates object where DAQ information is stored
            modelsDAQ, exitFlag = guiDAQ.select_daq_window(dt.modelsDAQ)
             [...]
```

Snippet 3.4.3 Interlinking between logic and GUI in function 'run_select_daq' with function 'select_daq_window' in './src/app/appDAQ.py'

Central to this architecture are two foundational objects. The first object is instantiated at the application's outset, entwined with the chosen DAQ device (Snippet 3.4.4). The second object emerges during the calibration setup phase (Snippet 3.4.5). While these objects maintain their distinct domains, they are engineered to function together, orchestrating the flow of data and control throughout the application.

```
[...]

# creates object where DAQ information is stored
            modelsDAQ, exitFlag = guiDAQ.select_daq_window(dt.modelsDAQ)
            # DAQ initiation with its corresponding model
            niDAQ = dt.niDAQ(modelsDAQ, exitFlag)
            if not exitFlag:
                niDAQ.initiate_daq()
            return niDAQ

[...]
```

Snippet 3.4.4 Creation of object 'niDAQ' in 'run_select_daq' function in './src/app/appDAQ.py'

```
[...]

            match method:
                case 'TEMP_VOLTAGE':
                    calibration =
appTempVoltCalibrate.run_temp_volt_calibrate(niDAQ)
                    if calibration is not None:
                        niDAQ.add_calibration_to_log(calibration)
                        niDAQ.set_calibration(repr(calibration))
                case 'EXPRESSION_INPUT':
                    calibration =
appExpressionInputCalibrate.run_expression_input_calibrate(niDAQ)
                    if calibration is not None:
                        niDAQ.add_calibration_to_log(calibration)
                        niDAQ.set_calibration(repr(calibration))
                case 'ACQUIRE_DATA':
                    if niDAQ.is_calibration_set:
                      break

[...]
```

Snippet 3.4.5 Object 'calibration' creation given the method chosen in function 'run_calibrate' in './src/app/appCalibrationMethod.py'

This architectural design optimizes the division of labor and ensures clear demarcations between components while promoting efficient collaboration. The logical and GUI-based components are interconnected, collectively working towards the application's overarching goals.

In essence, the application's high-level architecture exemplifies a structured yet interconnected framework, utilizing both logical and GUI features to cooperatively guide users through phases. The appendix, a repository of detailed explanations, will further clarify the interplay between code segments and components, providing a comprehensive understanding of the application's architectural intricacies.

3.5. BUILDING THE GRAPHICAL USER INTERFACE

The construction of the GUI adheres to a defined structure that fosters consistency and simplicity across different sections. Each GUI section is synchronized with its corresponding logical counterpart, guaranteeing integration of functionality and user interaction.

In a nutshell, the GUI structure follows a systematic pattern across all sections, ensuring uniformity and ease of use. This pattern encompasses the delineation of a layout (Snippet 3.5.2 and Snippet 3.5.3), the association of this layout with a designated window (Snippet 3.5.1), and the establishment of window behaviors within a recurring loop (Snippet 3.5.4). This cohesive structure serves as the foundation upon which the entire GUI framework is built.

```
[...]
    # launches window where the user can input calibration expression
    window, fig, figure_canvas_agg =
guiExpressionInputCalibrate.expression_calibrate_window()
    [...]
    calibration =
guiExpressionInputCalibrate.expression_input_calibrate_window_behavior(niDAQ, window,
fig, figure_canvas_agg)
[...]
```

Snippet 3.5.1GUI window structure in
'run_expression_input_calibrate' function in
'./src/app/appExpressionInputCalibrate.py'

```
[...]

return gt.gui_window_with_graph('Input Sensor Calibration Equation', layout,
gt.FIG_SIZE_WIDTH, gt.FIG_SIZE_HEIGHT, False)

[...]
```

Snippet 3.5.2 Return of 'expression_calibrate_window' in
'./src/gui/guiExpressionInputCalibrate.py'

```
[...]

def gui_window_with_graph(title, layout, figSizeWidth, figSizeHeight, isModal):
    """
    Initializes a PySimpleGUI window with a matplotlib using a CANVAS with empty
graph that can be updated later
    :param title: title of the window
    :param layout: layout designed for the window
    :param figSizeWidth: desired width of the graph
    :param figSizeHeight: desired height of the graph
    :param isModal: bool if window is modal
    :return: window, fig, figure_canvas_agg
    """
    [...]

    return window, fig, figure_canvas_agg

[...]
```

Snippet 3.5.3 Layout and window configuration returned in function 'gui_window_with_graph' in './src/guiTools.py'

```
[...]

def expression_input_calibrate_window_behavior(niDAQ, window, fig,
figure_canvas_agg):
    """
    Behaviour for direct expression input calibration window
    :param niDAQ: object
    :param window: pysimplegui window
    :param fig: calibration plot
    :param figure_canvas_agg: canvas for calibration plot
    :return: calibration object
    """
    calibration = ct.LinearCalibration()
    while True:
        event, values = window.read()

[...]
```

Snippet 3.5.4 Window Behavior loop in function 'expression_input_calibrate_window_behavior' in './src/gui/guiExpressionInputCalibrate.py'

The nature of this structure is chosen by its compatibility with PySimpleGUI, which accommodates the design. By maintaining consistency across GUI sections, the application establishes a sense of continuity, enabling users to navigate different sections easily.

The integration of relevant widgets plays a pivotal role in ensuring user engagement and interaction. Users' decisions are effectively captured through elements such as combo boxes, such as offering a dropdown menu to select the appropriate DAQ device (Snippet 3.5.5 and Fig. 3.5.1). The interface reflects user preferences, as the application adapts to the selected choice.

```
[...]
      [sg.Text('Select the model of the National Instruments DAQ:', pad=((0, 0),
(15, 0)))],
       [sg.Combo(modelDAQ,
                 default_value="Select the model...",
                 key='-MODEL-',
                 expand_x=True,
                 tooltip='Select an option before moving forward')],
[...]
```

Snippet 3.5.5 Combo box in layout to select DAQ model in 'select_daq_window' function in './src/gui/guiDAQ.py'

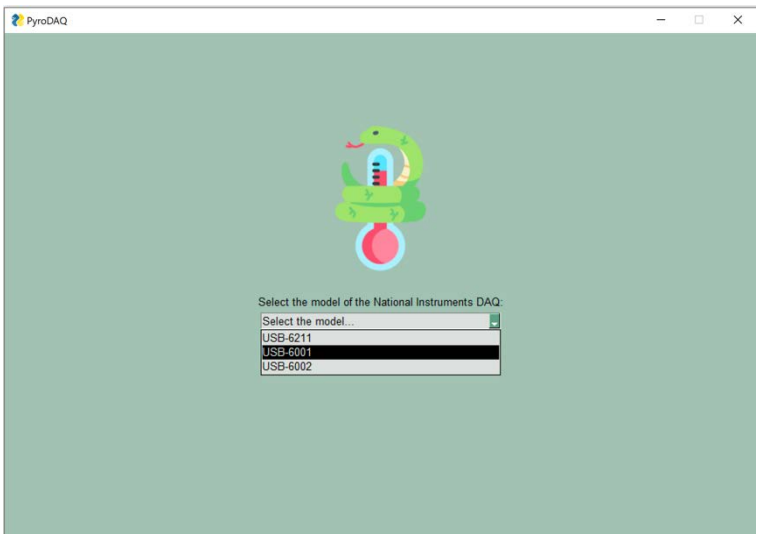

Fig. 3.5.1 Combo box shown as dropdown menu in DAQ selection window

Other elements like radio buttons further augment the user's control over the application's functionalities, like offering choices that align with the calibration process (Snippet 3.5.6, Snippet 3.5.7, Fig. 3.5.2 and Fig. 3.5.3). By selecting specific calibration options, users steer the application's behavior in line with their requirements.

```
[...]
                    [sg.Radio(gt.TEMP_VOLT_LIN_EQ,
                              group_id='exp_type',
                              default=True,
                              k='-LINEAR_EQ-',
                              enable_events=True,
                              pad=((10, 0), (10, 0)))],
                    [sg.Radio(gt.TEMP_VOLT_LEAST_SQUARES,
                              pad=((40, 0), 0),
                              group_id='lin_eq',
                              default=True,
                              enable_events=True,
                              k='-LEAST_SQUARES-')],
                    [sg.Radio(gt.TEMP_VOLT_LIN_INTERP,
                              pad=((40, 0), 0),
                              group_id='lin_eq',
                              default=False,
                              enable_events=True,
                              k='-LINEAR_INTERPOLATION-'),
[...]
```

Snippet 3.5.6 Radio elements for equation type in layout in function 'temp_volt_calibrate_window' in './src/gui/guiTempVoltCalibrate.py'

```
[...]

        if event == '-LINEAR_EQ-':
            gt.set_disabled(window, False, '-LEAST_SQUARES-', '-LINEAR_INTERPOLATION-')
            [...]
                if event == '-NON_LINEAR_EQ-':
            gt.set_disabled(window, True, '-LEAST_SQUARES-', '-LINEAR_INTERPOLATION-')

[...]
```

Snippet 3.5.7 Enabling and disabling radio buttons given the button selected in 'temp_volt_calibrate_window_behavior' function in './src/gui/guiTempVoltCalibrate.py'

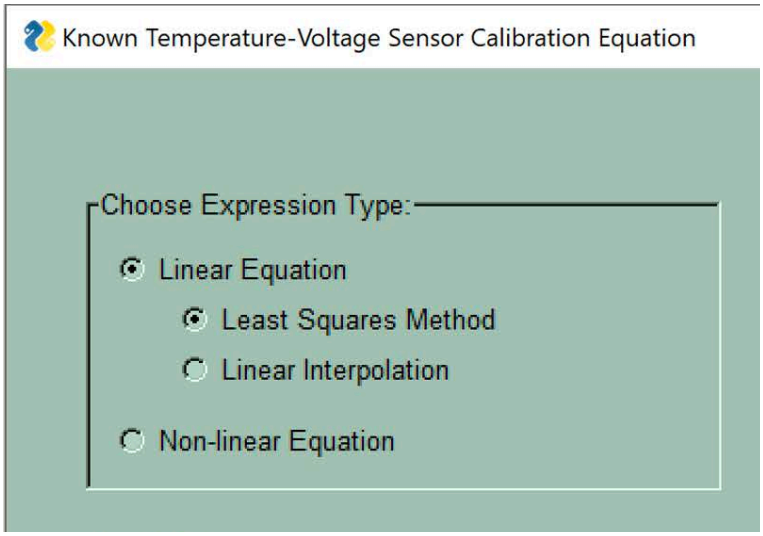

Fig. 3.5.2 Radio Button when 'Linear Equation' is selected, and all is enabled

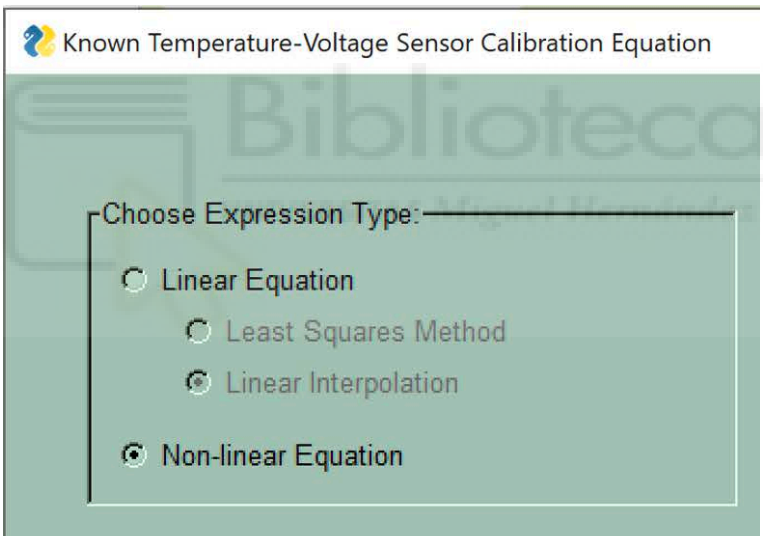

Fig. 3.5.3 Radio Button when 'Non-linear Equation' is selected, and options are disabled

The incorporation of user input is streamlined through input fields such as those that cater to voltage input (Snippet 3.5.8, Snippet 3.5.9 and Fig. 3.5.4). This interactive element enables users to input specific voltage values, an essential step in the calibration and data acquisition processes.

```
[...]

                [sg.Text('V =', k='-V_TXT-', pad=(10, 0)),
                   sg.Input(size=gt.SIZE_INPUT,
                            key='-V_INPUT-',
                            enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
                    sg.Text('T ='),
                    sg.Input(size=gt.SIZE_INPUT, key='-T_INPUT-',
enable_events=True),

                    sg.Button('Enter', k='-ENTER-', bind_return_key=True, pad=((10,
0), (10, 10)))]

[...]
```

Snippet 3.5.8 Voltage and temperature input in layout in 'temp_volt_calibrate_window' function in './src/gui/guiTempVoltCalibrate.py'

```
[...]

        if event == '-ENTER-':
           try:
               if values['-T_INPUT-'] == "":
                   raise ValueError("Values must be assigned")
               elif not gt.is_number(values['-T_INPUT-']):
                   raise ValueError("Values must be a numeric value.")

               if not window['-TOGGLE-'].metadata:
                   if values['-V_INPUT-'] == "":
                       raise ValueError("Values must be assigned")
                   elif not gt.is_number(values['-V_INPUT-']):
                       raise ValueError("Values must be a numeric value.")
                   inputValues = [float(values['-V_INPUT-']), float(values['-
T_INPUT-'])]
               else:
                   inputValues = [niDAQ.read_voltage(), float(values['-T_INPUT-'])]

[...]
```

Snippet 3.5.9 Voltage and temperature sequence in 'temp_volt_calibrate_window_behavior' function in './src/gui/guiTempVoltCalibrate.py'

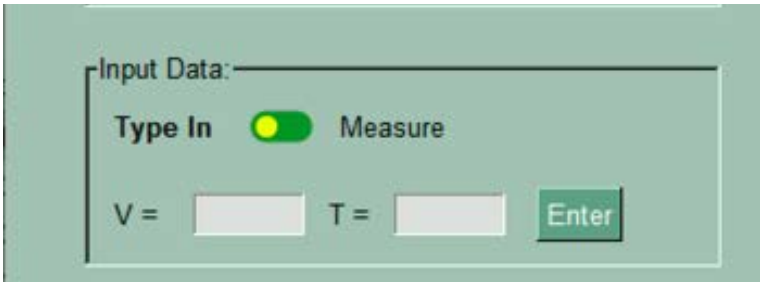

Fig. 3.5.4 Voltage and temperature inputs in known temperature-voltage window

The application's visualization of data handling, including logging and manipulation, is facilitated through intuitive data tables, and accompanying buttons such as clear and delete functions (Snippet 3.5.10, Snippet 3.5.11 and Fig. 3.5.5). The interactive table empowers users to observe logged data while retaining the ability to manage and edit entries.

```
[...]

            [sg.Table(values=[],
                      headings=['Voltage (V)', 'Temperature (ºC)'],
                      k='-TABLE-',
                      enable_click_events=True,
                      enable_events=True)],
             [sg.Text('Number of Samples: '),
              sg.Text('0', k='-N_SAMPLES-'),
              sg.Push(),
              sg.Button('Clear', k='-CLEAR-', tooltip=" Clear table ",
disabled=True),
              sg.Button('Delete', k='-DELETE-', tooltip=" Delete last row
", disabled=True)]

      [...]
```

Snippet 3.5.10 Data table, Delete and Clear buttons in layout in 'temp_volt_calibrate_window' function in './src/gui/guiTempVoltCalibrate.py'

```
[...]

      if event == '-DELETE-':
            del calibration[-1]
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=False)

        if event == '-CLEAR-':
            calibration.clear_data()
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=False)

[...]
```

Snippet 3.5.11 Delete and Clear behavior in 'temp_volt_calibrate_window' function in './src/gui/guiTempVoltCalibrate.py'

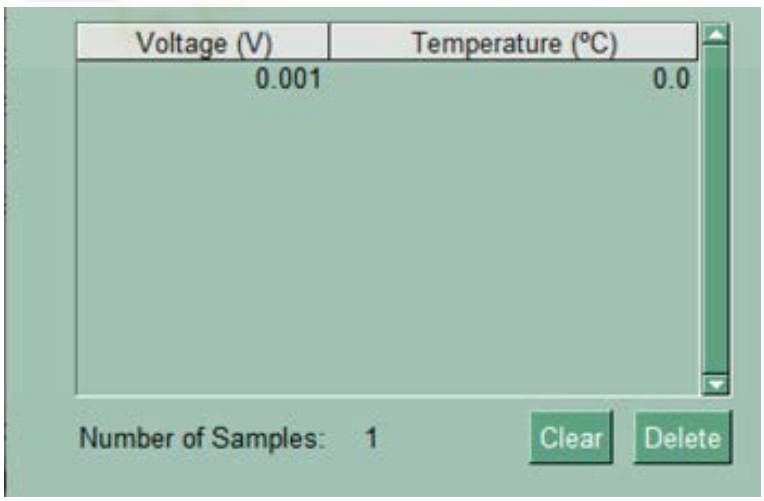

Fig. 3.5.5 Data table, Clear and Delete buttons in known temperature-voltage window

For effective data visualization, the application incorporates plot widgets that dynamically display acquired data (Snippet 3.5.12, Snippet 3.5.13 and Fig. 3.5.6). This

visual representation offers users an immediate grasp of data trends and patterns, enhancing their comprehension of temperature variations.

```
[...]
      [sg.Canvas(k='-CANVAS-', size=(200, 200))],
[...]
```

Snippet 3.5.12 Canvas for the plot in layout in 'temp_volt_calibrate_window' function in './src/gui/guiTempVoltCalibrate.py'

```
[...]
          calibration.update_figure(fig, figure_canvas_agg, known_expression=False)
[...]
```

Snippet 3.5.13 Canvas update for the plot in 'temp_volt_calibrate_window_behavior' function in './src/gui/guiTempVoltCalibrate.py'

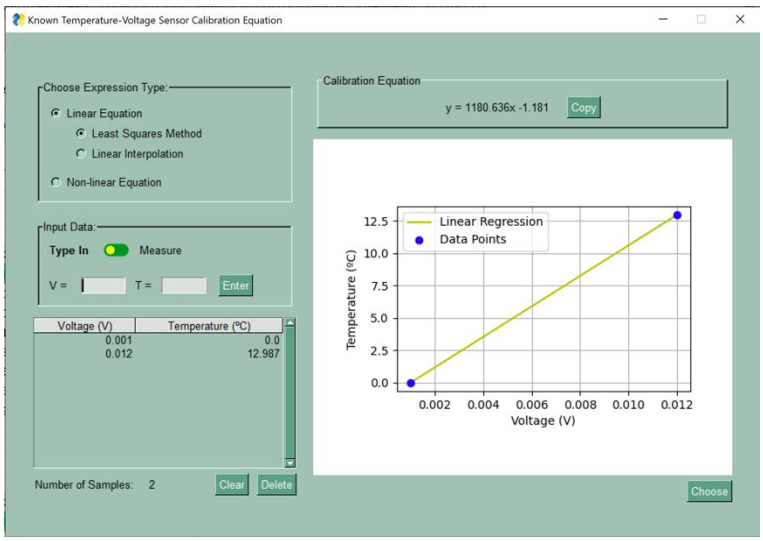

Fig. 3.5.6 Plotted data table in known temperature-voltage window

In summary, the process of building the GUI revolves around a structured approach that unifies the user experience across different sections. A consistent framework ensures coherence and familiarity, while the strategic incorporation of diverse widgets caters to user decisions, input, data handling, and visualization. This meticulous integration of GUI elements forms the bedrock of the application's user interface, creating an intuitive and engaging platform for users within the temperature-sensing circuit control context.

3.6. MANAGING USER INTERACTION AND CONTROL LOGIC

The basic principle for managing user interaction is to maximize usability by matching functionalities with their intended purpose. This approach streamlines the user experience by selectively enabling or disabling certain actions, ensuring that users are presented with pertinent options tailored to their current context.

For instance, the user interface is thoughtfully structured to reflect the logic that specific functionalities are only accessible when all relevant parameters have been accurately inputted. An illustrative instance involves the button labeled "Choose", as described earlier, by initially disabling it and subsequently activating it when all prerequisites are met, the application guides users along a structured path, fostering efficient decision-making.

Furthermore, the strategy extends to the user input process, particularly in situations where only numerical values are valid. Rather than permitting any input and subsequently notifying users of invalid entries, the application proactively filters out non-numeric characters as users type, preventing erroneous inputs from occurring (Snippet 3.6.1). This proactive measure enhances workflow, ensuring that users remain within the realm of valid inputs, thereby reducing the likelihood of errors.

```
def filter_numeric_characters(window, values, event, text_input_keys: list):
    """
    Filters out non-numeric text inputs so that even if the user types
letters and numbers, only numbers, '.' and '-'
    are shown
    :param window: window from gui where text is inputted and shown
    :param values: list of values in gui window
    :param text_input_keys: list with text-input keys
```

```
    :param event: event in gui window
    """
    # assigns text input key where there is an event
    k_event = text_input_keys[text_input_keys.index(event)] if
len(text_input_keys) > 1 else text_input_keys[0]
    # empty string where filtered out characters will be added
    filtered_chars = []
    # flag to signal if a '.' has already been typed in
    dot_found = False
    for char in values[k_event]:
        # adds if char is between 0-9
        if char.isdigit():
            filtered_chars.append(char)
        # adds '.' if it's the first one found
        elif char == '.' and not dot_found:
            filtered_chars.append(char)
            dot_found = True
        # adds '-' if it's in the first position
        elif char == '-' and len(filtered_chars) == 0:
            filtered_chars.append(char)

    values[k_event] = ''.join(filtered_chars)
    window[k_event].update(values[k_event])
```

Snippet 3.6.1 'filter_numeric_characters' function in './src/guiTools.py'

The intermediary control layer between the GUI and Python code functions as a crucial bridge, orchestrating communication, and command flow. These intermediary functions, positioned at each juncture of the application's process, facilitate coordination between the GUI and underlying logic. Their role encompasses the initiation of essential tasks, ensuring that subsequent steps are executed in a controlled and sequential manner.

This separation of layers proves indispensable, particularly when handling multiple windows. By avoiding overlaps and establishing a sequential workflow, the application

sustains its coherence and functional integrity. This layer serves as a conduit, ensuring that each GUI action corresponds with the appropriate command on the logic side.

This symbiotic relationship extends further as GUI functions operate as intermediaries between user input and logical commands. Upon completion of a GUI process, such as selecting a calibration method, the resulting action is transmitted back to the logical layer. This can occur through the return from a function or by updating pertinent objects. This tightly intertwined interaction ensures an exchange of information, aligning user input with the subsequent logical steps.

The upcoming sections delve deeper into the intricacies of control logic within the context of interfacing with the DAQ and the temperature sensor. These discussions give detailed insights into the management of user interface and control logic and further clarify the sequencing of the application.

The philosophy governing user interaction management within the application embodies a systematic approach that aligns functionalities with user context. By proactively limiting invalid inputs, controlling user access to functionalities, and utilizing intermediary control layers, the application orchestrates a user-friendly and coherent experience. This methodology is essential in seamlessly connecting the GUI with underlying logic, fostering a harmonious synergy between user interactions and the application's overarching goals.

3.6.1. INTEGRATING NATIONAL INSTRUMENTS DAQ

The integration of the NI DAQ device using NI-DAQmx is the fundamental aspect of the application's functionality. This integration entails distinct tasks, primarily focused on facilitating analog channel output and input operations.

At the heart of the integration lies the utilization of the DAQ to accomplish essential tasks: writing and reading. The writing task involves channeling a set output analog voltage to power the circuit. The powering of the circuit will be further discussed in Section “EXPERIMENTAL SETUP," where the specifics of the power source and its implications will be elaborated upon.

This function operates in the background, automatically activated when the circuit is deemed correctly set up and the user initiates temperature reading (Snippet 3.6.2). The reading task, on the other hand, is indirectly prompted by the user when they opt to acquire data or measure voltage (Snippet 3.6.3).

```
def run_data_acquisition(niDAQ):
    """
    Runs data acquisition
    :param niDAQ: object where data will be stored
    :return:
    """
    # launches window where the user can input calibration expression
    window, fig, figure_canvas_agg =
guiDataAcquisition.data_acquisition_window(niDAQ.calibration)
    niDAQ.set_task_start(1)
    niDAQ.set_task_write(1)
    guiDataAcquisition.data_acquisition_window_behavior(niDAQ, window, fig,
figure_canvas_agg)
    niDAQ.set_task_stop(1)
```

Snippet 3.6.2 Writing task in function 'run_data_acquisition' in './src/app/appDataAcquisition.py'

```
        def read_voltage(self):
        """
        Simulates the reading of the voltage by the DAQ

        returns:
            voltage (float): reading of voltage by the DAQ
        """
        self.set_task_start(0)
        match self.model:
            [...]

             case 'USB-6001':
                # simulation of temperature reading by the DAQ
                voltage = self.task_ai_ao[0].read()

            [...]
        self.set_task_stop(0)
        return round(voltage, 3)
```

Snippet 3.6.3 Reading task in 'read_voltage' function in './src/daqTools.py'

A foundational aspect of this integration is the application's capacity to recognize the presence of a connected DAQ. Through a function, the program detects whether the DAQ is properly connected (Snippet 3.6.4, Snippet 3.6.5, Snippet 3.6.6 and Fig. 3.6.1). This preemptive check serves as a valuable precaution, reminding users to connect the DAQ and avoid errors stemming from unintentional omissions.

```
def is_daq_connected():
    system = nidaqmx.system.System.local()
    devices = system.devices
    return len(devices) > 0
```

Snippet 3.6.4 'is_daq_connected' function in './src/app/daqTools.py'

```
    def set_tasks(self):
        if is_daq_connected():
            for channel in range(2):
                self.task_ai_ao.append(nidaqmx.Task())
        else:
            raise ValueError("Number of devices found in system is 0")
```

Snippet 3.6.5 Check if DAQ is connected in 'set_tasks' function in './src/app/daqTools.py'

```
def run_select_daq():
    """
    Runs daq selection
    :return:
    """
    while True:
        try:
            [...]
        except ValueError as e:
            guiDAQ.no_daq_detected_popup(e)
```

Snippet 3.6.6 Catching the no DAQ error in 'run_select_daq' function in './src/app/appDAQ.py'

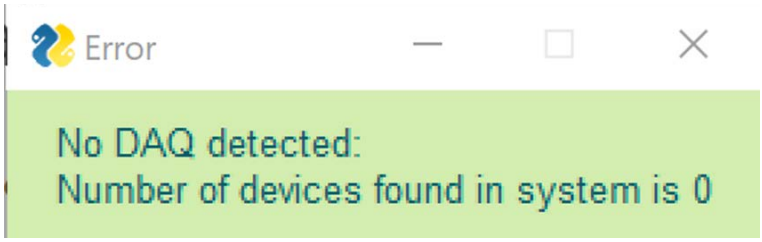

Fig. 3.6.1 Popup message when there is no DAQ detected

Upon selecting the suitable DAQ model through the user's action and choice, as stated before, an object is initialized with the relevant model information. This object serves as the central focal point of the application. Once calibration is assigned to this object, it becomes the source from which all functions essential for data addition, management, and trial-related information stem.

Further details regarding the decision-making process behind the selection of the DAQ model are elaborated upon in the section titled " SUGGESTIONS FOR FUTURE RESEARCH AND DEVELOPMENT".

3.6.2. INTEGRATING TEMPERATURE SENSING AND CIRCUIT CONTROL FUNCTIONALITY

The integration of temperature sensing functionality into the application centers around the circuit control capabilities established in the preceding section "Integrating National Instruments DAQ." This symbiotic relationship empowers the application to both power and read the circuit, thereby facilitating temperature-sensing processes.

In the calibration process, as noted earlier, a pivotal object is instantiated, serving as a central hub for all calibration-related procedures. This object becomes the focal point for all endeavors related to calibration setup, acting as a hub from which various trials and configurations emanate. This approach maintains a separation of concerns, allowing the two distinct objects to function concurrently while delaying interaction until the user selects a specific calibration (Snippet 3.6.7). This design choice provides users with the autonomy to experiment with different calibrations while retaining the freedom to opt for previously calculated values.

```
[...]

          case 'TEMP_VOLTAGE':
                   calibration = appTempVoltCalibrate.run_temp_volt_calibrate(niDAQ)
                   if calibration is not None:
                       niDAQ.add_calibration_to_log(calibration)
                       niDAQ.set_calibration(repr(calibration))

[...]
```

Snippet 3.6.7 Assignation of calibration to niDAQ object after 'run_temp_volt_calibrate' has run, in 'run_calibrate' function in './src/app/appCalibrationMethod.py'

3.7. USER INTERACTION FLOW

The subsequent section provides a comprehensive walkthrough of the user interaction flow, delineating the sequence of actions from the initiation of the application to the culmination of data analysis. This step-by-step exploration sheds light on the user's journey, underscoring the logical progression and transitions that characterize the user experience.

1. *Launching the Application and GUI Initialization*: The user launches the Python-based application designed for controlling the NI DAQ. Upon launch, the graphical user interface (GUI) of the application is presented to the user.

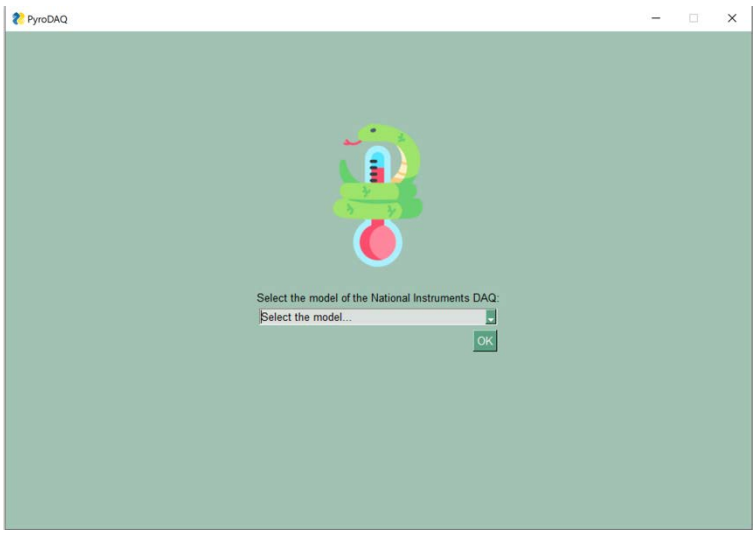

Fig. 3.7.1 DAQ model selection window

2. *DAQ Model Selection:* The user is provided with options from a dropdown menu to select the appropriate model of the NI DAQ.

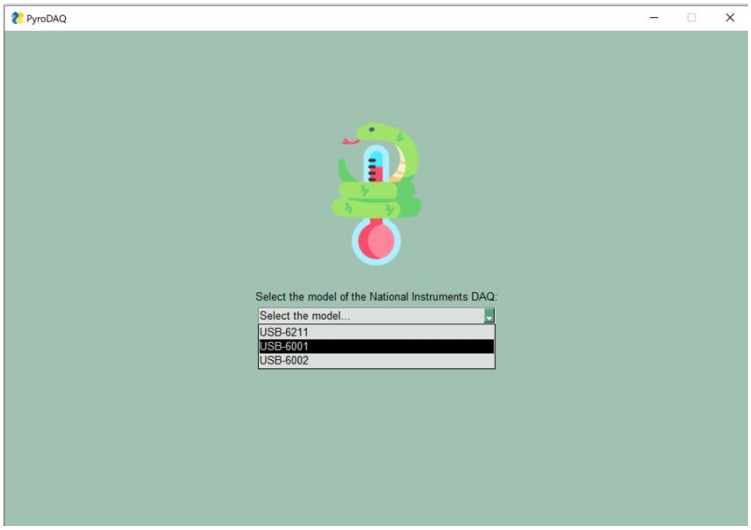

Fig. 3.7.2 DAQ model selection with options window

3. *Calibration Configuration:* At this stage, the user is guided through the process of setting the calibration value for the temperature-sensing circuit. They are prompted to select the desired calculation method. Two options are provided:

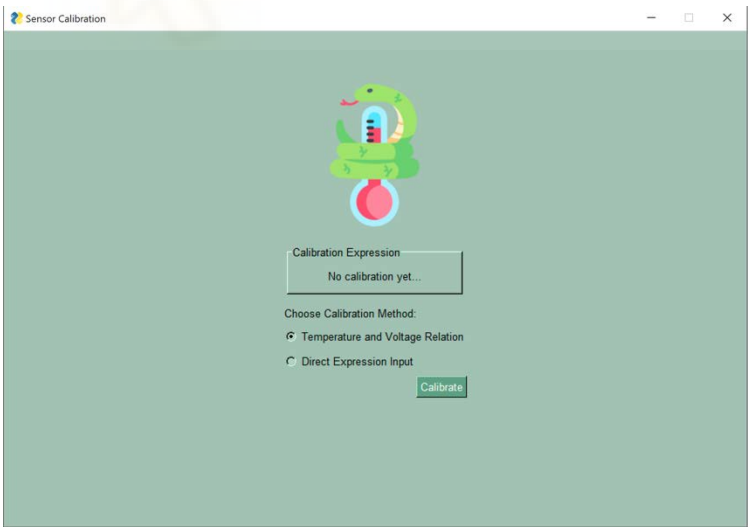

Fig. 3.7.3 Calibration method window

a. *Known Temperature-Voltage Value:* Users are presented with the choice between calculating a linear or non-linear equation.

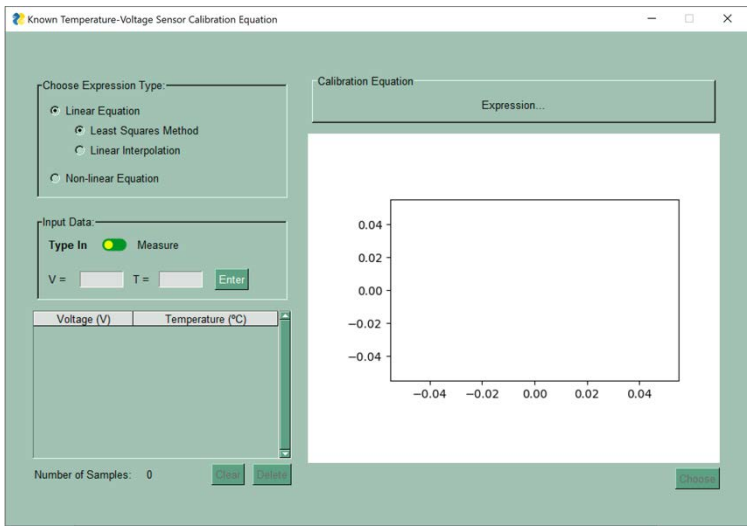

Fig. 3.7.4 Equation type choices

If the linear option is chosen, they can further decide between the least squares method or point interpolation. When two data points are collected, users are prompted to choose two points for interpolation. The user's choice establishes the expression type.

Next, users are presented with options regarding the voltage value. They can manually input the voltage or allow the DAQ to measure the voltage at that instant. Correspondingly, the user inputs the temperature value and, if necessary, the voltage. All collected data is visually displayed in a table format, providing options to delete the last added point or clear the table.

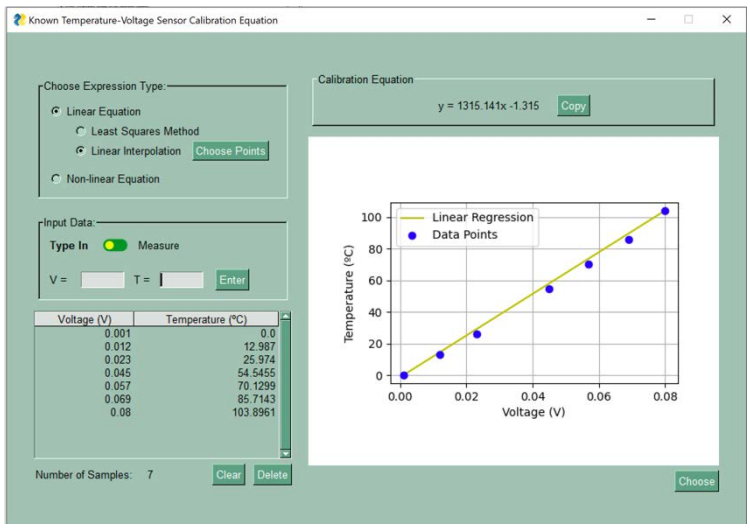

Fig. 3.7.5 Linear Interpolation method with "Choose Points" button

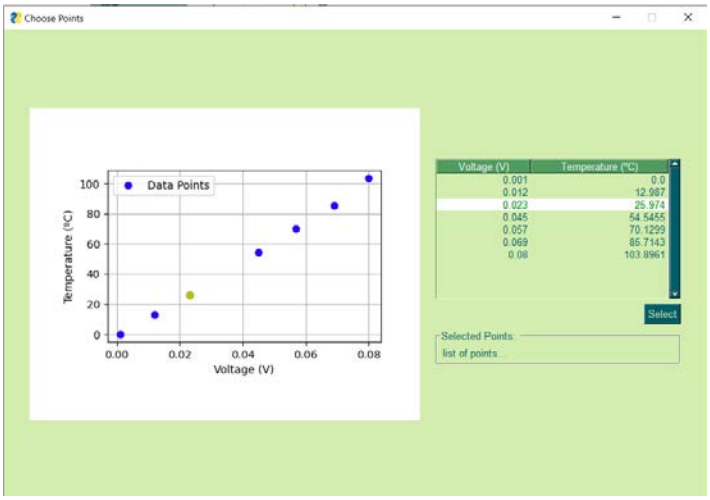

Fig. 3.7.6 Choose Points window, point clicked on

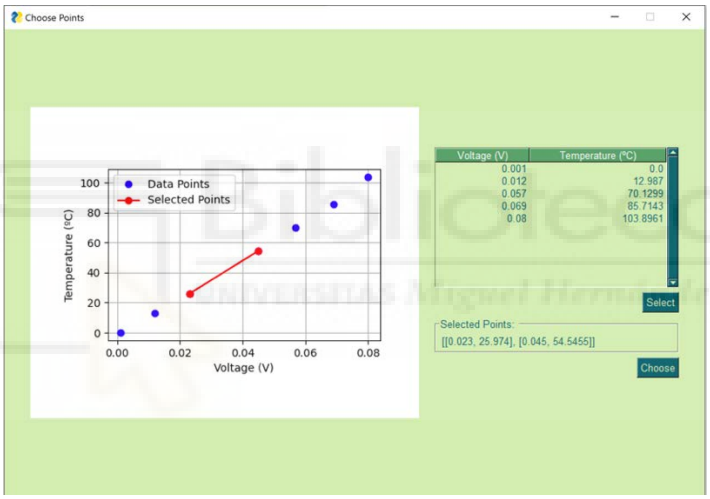

Fig. 3.7.7 Choose Points window, two points selected

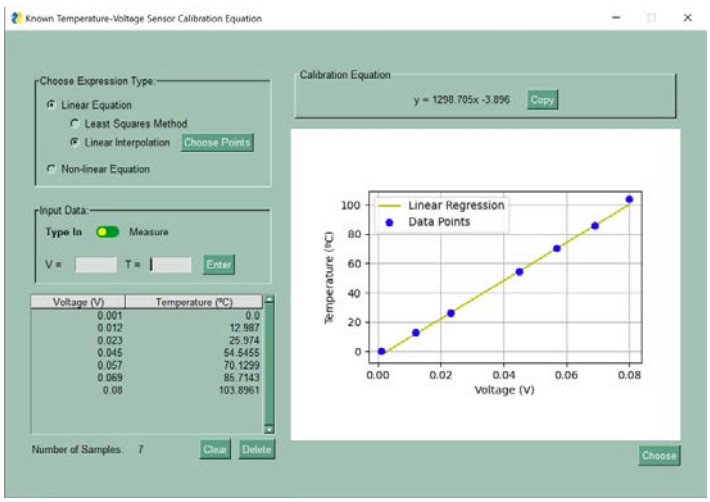

Fig. 3.7.8 Subsequent calibration from choosing points

Once either two or three data points are collected (depending on the expression type), the calculated expression is presented to the user. The user has the option to copy the expression if desired. Alongside the expression, both the data and the expression are plotted. Users can choose to change the expression calculation method while retaining the collected data points.

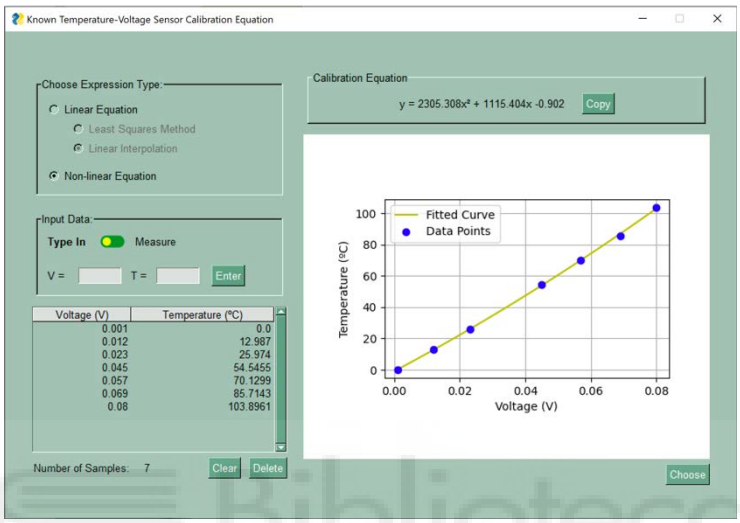

Fig. 3.7.9 Change to non-linear equation

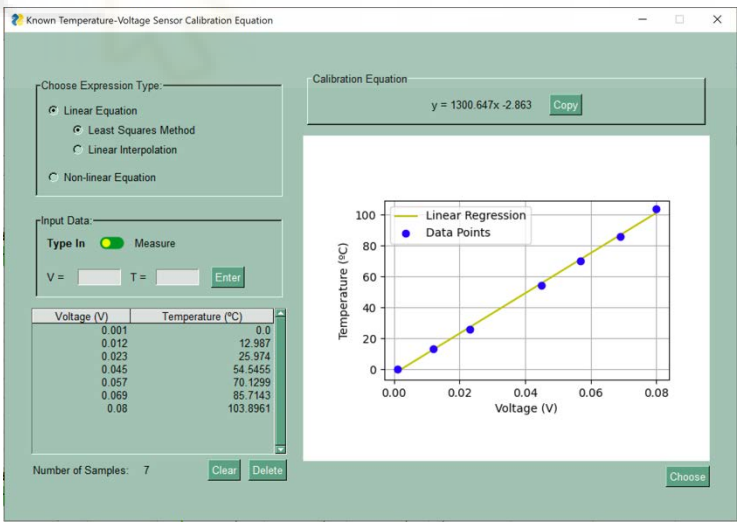

Fig. 3.7.10 Change to linear equation, least squares method

Upon satisfaction, users can proceed by selecting the "Choose" button, which redirects them to the calibration menu. Here, the selected calibration is set.

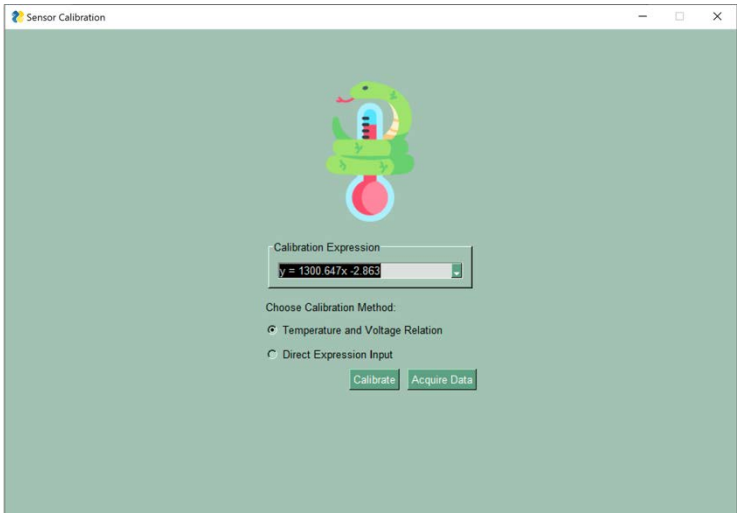

Fig. 3.7.11 Set Calibration

b. *Known Expression Parameters:* Users are given the option to choose between a linear or non-linear calibration expression. Depending on the choice, they need to input two or three parameters, respectively.
After setting the parameters, the expression is calculated and displayed alongside the corresponding plot. The option to copy the expression is also available.

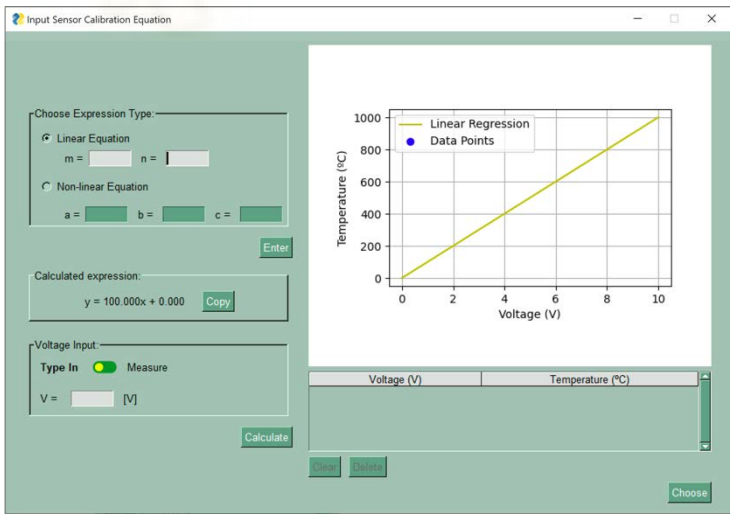

Fig. 3.7.12 Linear equation calibration input with corresponding plot

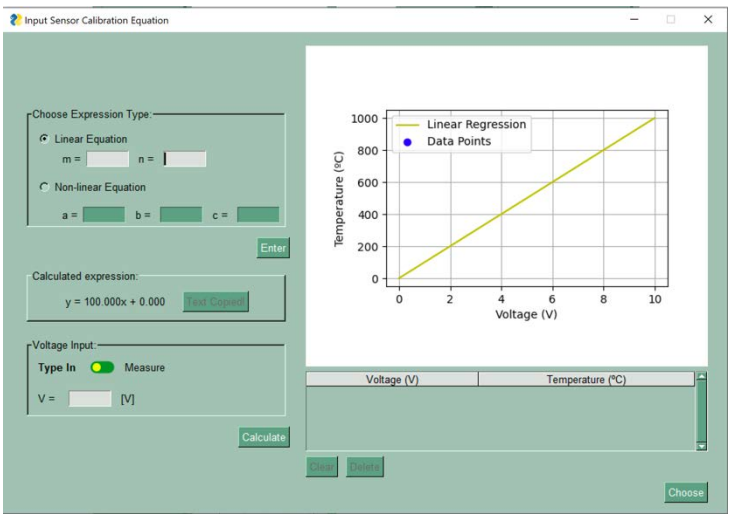

Fig. 3.7.13 Copy equation selected

Following this, users are prompted to select between typing in the voltage or measuring it instantly. A temperature point is calculated and appended to the data table for each voltage point. The table provides the capability to delete the last point added or clear the entire table. All data points are visualized on the plot, alongside the expression.

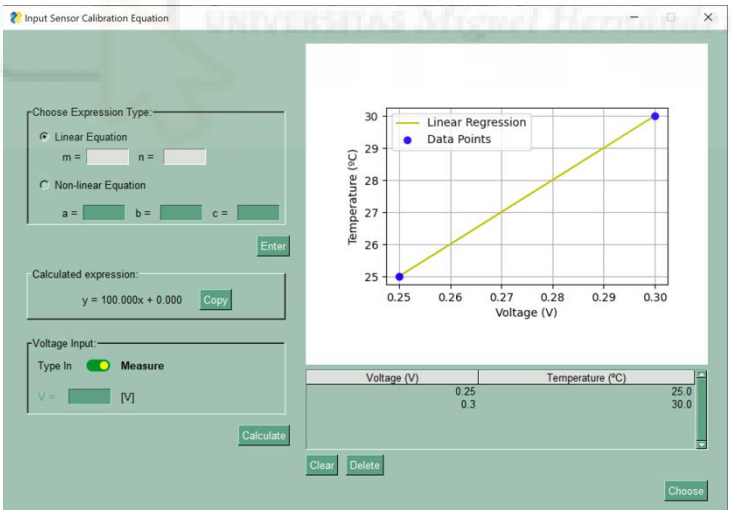

Fig. 3.7.14 Data points logged and represented in plot

Once users are content with the calibration setup, they can proceed to finalize their choice. They are then returned to the calibration method section, where the newly calculated calibration is set.

From this point onward, users are empowered to select from previously calculated calibrations, add new calibrations, or move forward to the data acquisition phase.

Fig. 3.7.15 Calibration log with saved calibrations and options to calibrate and acquire data

4. *Data Acquisition Setup and Display:* At this phase, the user engages in configuring parameters for data acquisition.

   The user is presented with the ability to incorporate alarms that monitor temperature values. Maximum and/or minimum temperature alarms can be added, with triggers set for temperatures surpassing or falling below specified values. These alarms are visually represented as LEDs that will turn on and off, as well as dashed lines on the plot, enabling users to intuitively identify potential issues. Users can edit or disable these alarms according to their requirements.

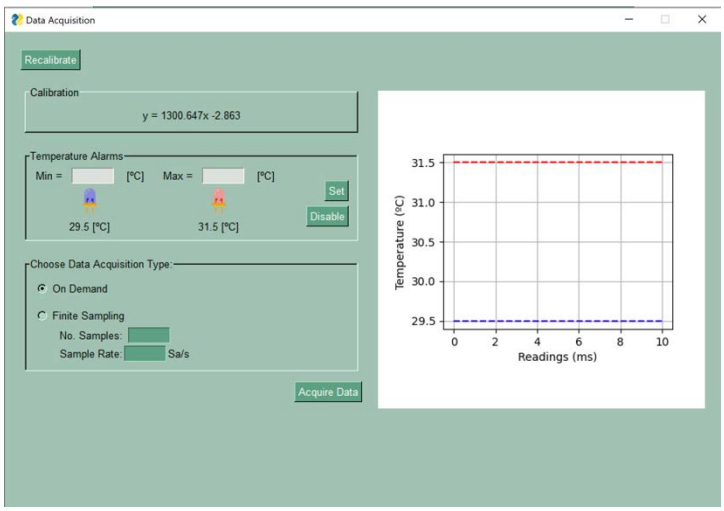

Fig. 3.7.16 Example of alarms being set at 29.5ºC and 31.5ºC

Users have the choice between two modes: on-demand data acquisition or finite sampling, using the calibration established in the preceding step.

a. *On-Demand Data Acquisition:* When this option is selected and "Acquire Data" clicked on, data acquisition commences automatically with a default time interval of 500ms.

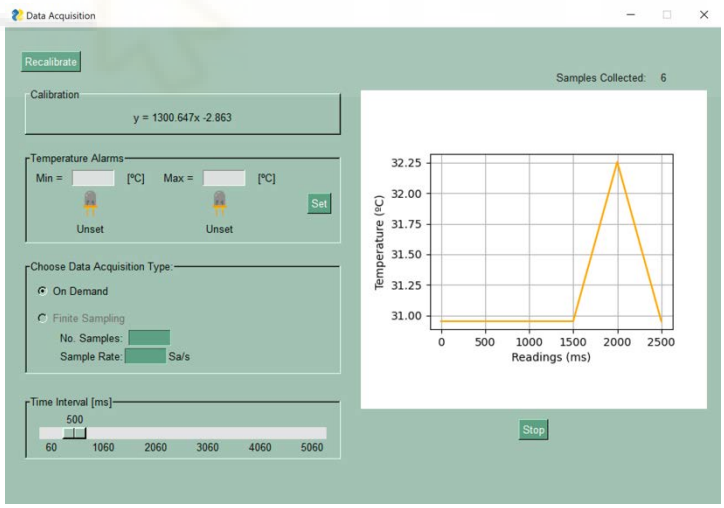

Fig. 3.7.17 On demand acquisition with 500ms time interval

Users can dynamically adjust this interval using a slider. Until the user initiates the "Stop" command, the acquisition persists. Data collected is graphically represented in a plot, accompanied by a tally of the acquired

samples. Users can opt to restart the process or modify the configuration as needed.

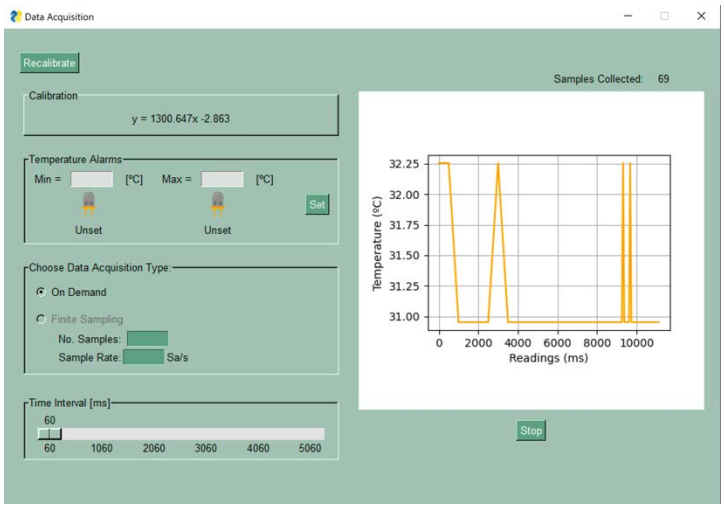

Fig. 3.7.18 On demand acquisition with 60ms time interval

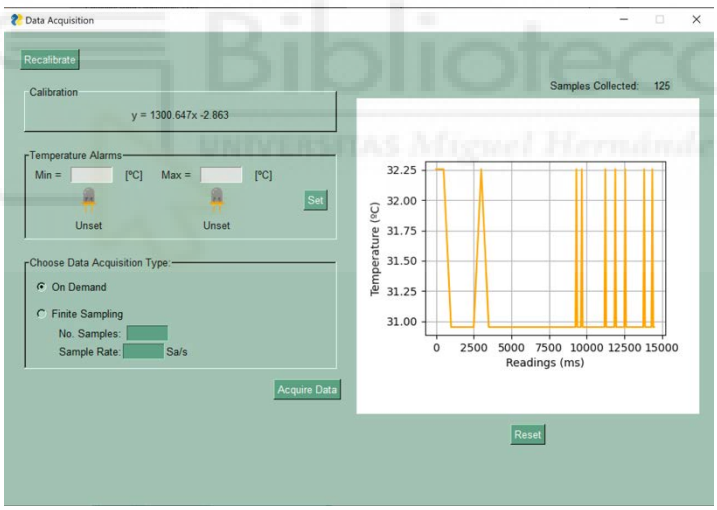

Fig. 3.7.19 On demand acquisition stopped manually

b. *Finite Sampling:* For this option, users input the desired number of samples along with the time interval between each sample. Upon activating the "Acquire Data" button, data collection begins. Like the previous mode, a sample counter and a graphical plot of the data are presented.

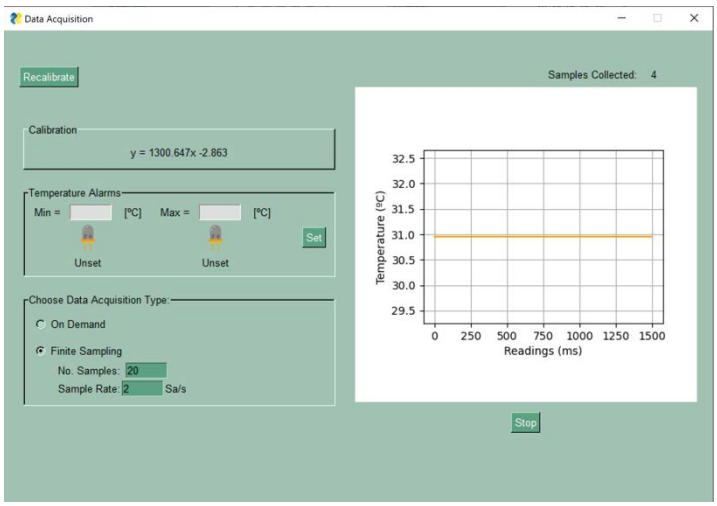

Fig. 3.7.20 Ongoing finite sampling acquisition with the parameters: 20 samples and 2 Sa/s

Unless the user triggers the "Stop" command, data collection continues until the targeted number of samples is achieved. Upon completion, the option to save the data emerges.

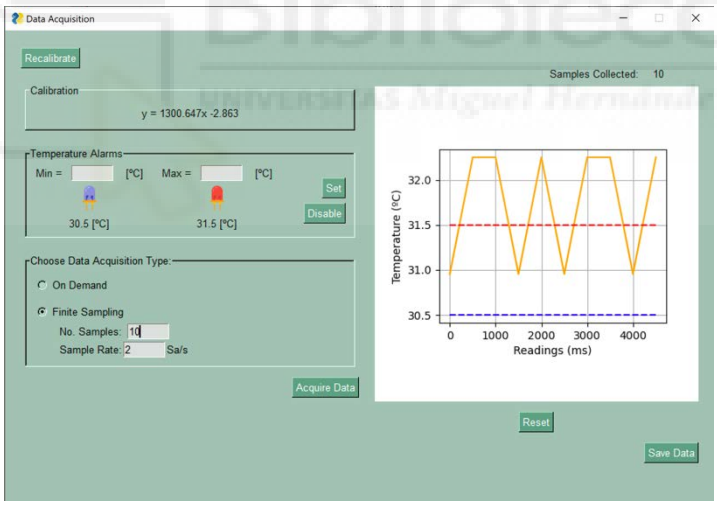

Fig. 3.7.21 finite sampling acquisition finished with the parameters: 10 samples and 2 Sa/s

If the user chooses to save the data, a prompt appears asking them to name the file and designate its location. This generated CSV file (Snippet 3.2.1) encompasses the DAQ information, the calibration established for the acquisition, pertinent parameters, the alarm log, and the gathered data points.

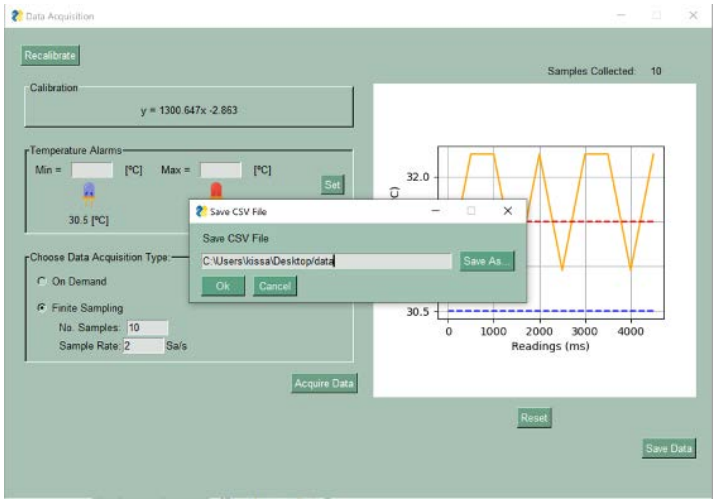

Fig. 3.7.22 Save data prompt

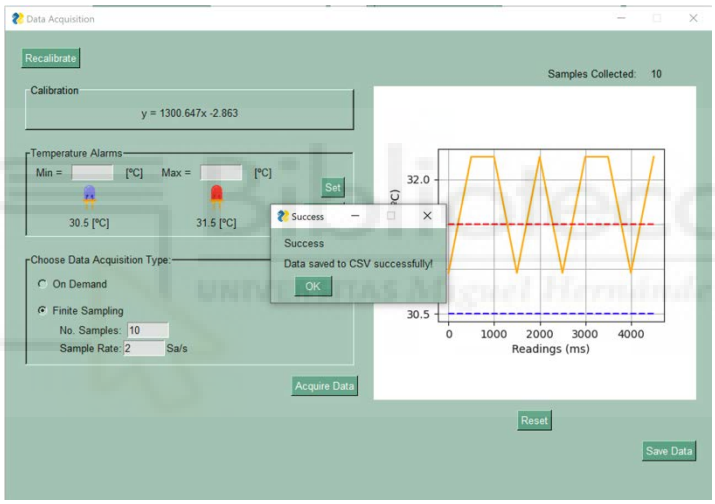

Fig. 3.7.23 Data successfully saved

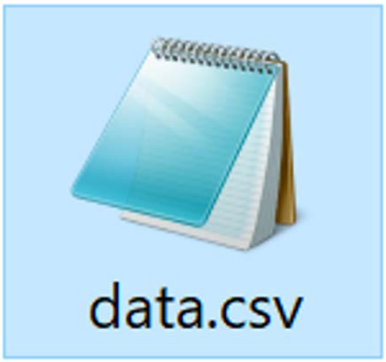

Fig. 3.7.24 Data successfully created

Upon completion of the data acquisition process, users are given the choice to either continue acquiring data or return to the calibration phase, facilitating recalibration as needed.

5. *Closing the Application:* Finally, from the calibration menu, the user can close the application when they're done.

3.8. ERROR HANDLING

The strategy employed for detecting and addressing errors within the application draws upon the basis of constraining user input, a principle discussed in earlier sections. This approach significantly minimizes the potential for errors and enhances the application's overall robustness, allowing for a more streamlined error-handling process.

By implementing measures like restricting user input to valid characters and numerical values, most potential errors are preemptively mitigated. This proactive stance ensures that input values are appropriate and relevant, lowering the possibility of errors originating from incorrect or inappropriate input.

For instance, as explained before, the application's user input mechanism is designed to exclusively accept numbers or pertinent symbols, effectively guaranteeing the accuracy of numeric inputs. This screening method significantly reduces the number of potential errors, leaving only a handful that require care.

Should an error be detected, the application uses a popup message mechanism to communicate the issue to the user, exposing the problem and guiding users toward rectification.

Among the remaining error scenarios, such as omitting necessary inputs or entering a minimum value greater than the maximum, must be considered. To address these scenarios, the application implements try-except blocks that intercept errors as they arise, preventing their escalation and potential disruption of user experience.

```
[...]

        if event == '-SET-':
            try:
                # checks if both inputs are empty
                if all(values[key] == "" for key in alarm_input_keys):
                    raise ValueError("Values must be assigned")
                elif all(values[key] != "" for key in alarm_input_keys):
                    alarm_min, alarm_max = gt.to_number_n_dec(gt.N_DECIMALS,
values['-MIN_TEMP_INPUT-'],
                                                              values['-
MAX_TEMP_INPUT-'])
                    if alarm_min >= alarm_max:
                        raise ValueError("Min alarm can't be bigger or equal to max
alarm")
                    niDAQ.set_alarm_min(alarm_min)
                    niDAQ.set_alarm_max(alarm_max)
                else:
                    if values['-MIN_TEMP_INPUT-'] != "":
                        [alarm_min] = gt.to_number_n_dec(gt.N_DECIMALS, values['-
MIN_TEMP_INPUT-'])
                        if niDAQ.is_alarm_max_set() and (alarm_min >=
niDAQ.get_alarm_max()):
                            raise ValueError("Min alarm can't be bigger or equal to
already set max alarm")
                        else:
                            niDAQ.set_alarm_min(alarm_min)
                    if values['-MAX_TEMP_INPUT-'] != "":
                        [alarm_max] = gt.to_number_n_dec(gt.N_DECIMALS, values['-
MAX_TEMP_INPUT-'])
                        if niDAQ.is_alarm_min_set() and (alarm_max <=
niDAQ.get_alarm_min()):
                            raise ValueError("Max alarm can't be bigger or equal to
already set min alarm")
                        else:
                            niDAQ.set_alarm_max(alarm_max)
                niDAQ.update_figure(fig, figure_canvas_agg)
                niDAQ.trigger_alarm_icon(window, alarm_icon_keys)
                gt.set_visible(window, True, '-DISABLE-')
            except Exception as e:
                sg.popup_error(str(e), title="Error")

[...]
```

Snippet 3.8.1 try-except block for setting alarms in function 'data_acquisition_window_behavior' in './src/gui/guiDataAcquisition.py'

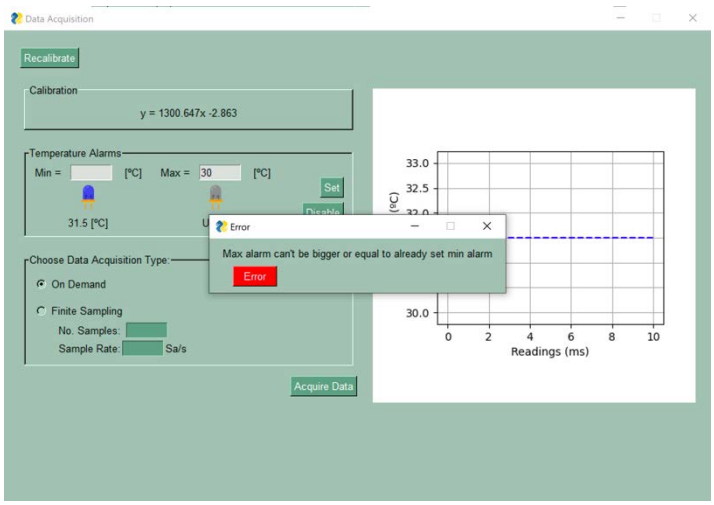

Fig. 3.8: Popup informing the user that there's an error setting max alarm

3.9. TESTING, DEBUGGING, AND VALIDATION

The comprehensive testing, debugging, and validation process, as outlined in Section 2.4 "TESTING AND QUALITY ASSURANCE STRATEGY", was implemented throughout the development journey. This methodical approach proved invaluable in uncovering numerous bugs and errors that were promptly addressed and resolved. While many issues were successfully handled, one emerged that resisted simple solutions.

During the examination of extreme scenarios within the system testing stage, a critical oversight of the hierarchical control structure of the NI DAQ came to light. The real sample rate of data acquisition was different from what it should be, which prompted a more thorough examination of the underlying system architecture.

The separation between control logic and the user interface holds vital importance, considering the simultaneous functioning of both components. Unfortunately, this principle was unintentionally overlooked, leading to a situation where the NI DAQ was erroneously managed within the user interface segment of the code.

This mistake had a notable outcome as a result: the update frequency of the NI DAQ became reliant on the frequency of the PySimpleGUI function. This linkage unintentionally limited the DAQ's sample rate, which had an impact on its ability to acquire data. Once this issue was identified, it was evident that considerable refactoring of the project's code was necessary to correct it.

By the time the oversight was identified, a substantial portion of the project had already been completed. The decision to proceed with the existing work was influenced by the nature of the application's operational scenarios, which were temperature-based. Given the relatively gradual changes in temperature, a hyper-precise frequency was not paramount, and the user experience remained relatively unaffected.

Nevertheless, it was acknowledged that not fully capitalizing on the DAQ's potential capabilities represented a significant compromise. The chosen solution involved limiting the frequency available for the user, enabling a balance between the existing system architecture and the user's requirements.

While this discovery posed a notable challenge, it also reinforced the importance of vigilance in system architecture design and highlighted the potential ramifications of decisions made at the outset of a project.

3.10. SUGGESTIONS FOR FUTURE RESEARCH AND DEVELOPMENT

Considering the limitations highlighted in the preceding section, several insightful suggestions for future research and development come to the forefront:

Firstly, addressing the challenge of separating the control of the DAQ presents an immediate opportunity for improvement. This could be effectively resolved by extracting the pertinent functions from the GUI component and integrating them in the control portion. By integrating these aspects, a more coherent and efficient control structure could be established, ensuring better synchronization between user interaction and DAQ functionality. This evolution could serve to enhance the overall performance and usability of the application.

Basic aesthetic improvements can be considered for the GUI, such as enhancing the increments on the x-axis to achieve a cleaner look, especially when dealing with larger time values where the current representation might appear less refined.

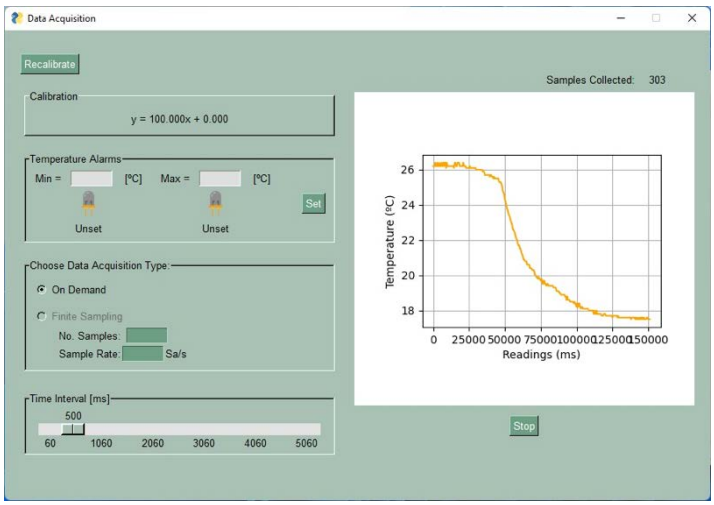

Fig. 3.10: GUI x-axis representation at high time values

It's important to note that this application represents a first version, and future iterations have the potential to build upon these foundational aspects, incorporating refinements and user-driven enhancements.

Furthermore, the prospect of extending the application's capabilities to accommodate the simultaneous utilization of two DAQ units holds potential. This expansion would provide users with enhanced flexibility in data acquisition scenarios that involve multiple sensors or circuits. Additionally, the feasibility of incorporating other models of DAQ units, beyond the initial three, is a noteworthy avenue for future exploration. The foundation for such integration is already laid through the existing option to choose among the three models.

This initial choice was introduced owing to the differences among the three DAQs tailored for this application's scope. The underlying revelation, however, surfaced subsequently; it became apparent that a unified codebase could be harnessed across these models, given the commonality in basic functionalities like reading and writing.

This foresight guided the decision to retain the choice among the existing models, as it was deduced that the foundational architecture already in place would readily facilitate the integration of further models in the future. The architectural flexibility laid the groundwork for seamless expansion, should the need arise to encompass additional DAQ variants.

In the realm of educational enrichment, a compelling direction involves integrating code inspection as an instructive tool. The open-source nature of Python and its widespread utilization within the field make it an invaluable resource for students. By enabling students to delve into the code behind the application, a deeper understanding of software-hardware interactions in the context of electronics can be fostered. This presents a unique opportunity for students to grasp the intricacies of the underlying mechanisms, which is often obscured in tools like LabVIEW.

Considering the enhancement of the student learning experience, the development of a mobile application could be a promising endeavor. Such an extension would bridge traditional learning methodologies with modern technology, providing students with a more engaging and immersive learning experience.

Lastly, for broader accessibility and improved student comprehension, incorporating translation options in languages such as Spanish or Valencian (as spoken where the project has been developed) could prove beneficial. This localization effort would ensure that a wider spectrum of students can engage with the application in a language that aligns with their comfort and familiarity.

These suggestions collectively offer a roadmap for refining and expanding the application's functionality, impact, and usability in both educational and practical contexts.

4. EXPERIMENTAL SETUP

In this section, we delve into the experimental setup, providing a comprehensive guide to the practical aspects of utilizing this application. This section offers a condensed overview of the experimental setup, focusing on the critical components and processes involved in harnessing the application's capabilities for temperature sensing and data acquisition. For a more detailed walkthrough, readers are encouraged to refer to the appended "PyroDAQ Student's Guide."

4.1. CIRCUIT SETUP

The central configuration employed in this experiment is a Wheatstone bridge, using a Pt100 as the temperature sensor. The Pt100, a platinum resistance temperature detector, exhibits a well-defined resistance-temperature relationship, rendering it highly suitable for temperature sensing applications. Pt100 probes operate on the principle of the (practically linear) change in resistance exhibited by a platinum wire as a function of temperature, which can be expressed as:

$$R(t) = R_0[1 + \alpha(t - t_0)$$

Equation. 1 Temperature and resistance variation

With $t_0 = 0°C, R_0 = 100\ \Omega,$ and $\alpha = 0{,}00385°C^{-1}$.

A Wheatstone bridge is a circuit configuration used for measuring electrical resistance with high accuracy [14]. It consists of four resistors arranged in a diamond-shaped configuration (as depicted in Fig. 4.1.1). The Wheatstone bridge operates on the principle of balancing two legs in the bridge circuit, with one leg having an unknown resistance to be measured (Rx).

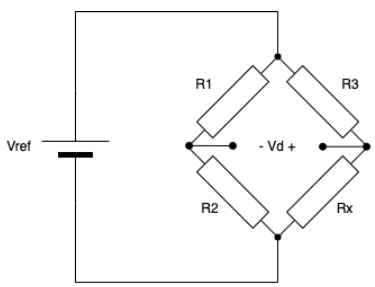

Fig. 4.1.1 Wheatstone bridge configuration

The voltage difference between the central points of the two legs is known as the "output voltage" or "bridge output." When the ratio $\frac{R1}{R2}$ is equal to the ratio $\frac{R3}{Rx}$, the bridge is said to be in a balanced or null state (Equation. 2). This means that there is no net current flowing through the central connection point, known as the bridge's "null" point. When a change in resistance occurs in any of the four resistors (in particular Rx) the bridge becomes unbalanced. This results in a voltage difference at the output that can be measured (Equation. 3).

$$\frac{R1}{R2} = \frac{R3}{Rx}$$

Equation. 2 Balanced Wheatstone bridge condition

$$V_d = V_{REF} \times \left(\frac{R_x}{R_2 + R_x} - \frac{R_3}{R_1 + R_3}\right)$$

Equation. 3 Output voltage equation for Wheatstone bridge

By precisely calibrating the bridge with known resistors and recording the output voltage, one can establish a relationship between resistance changes and the corresponding output voltage variations. This calibration allows you to use the Wheatstone bridge for accurate measurements of unknown resistances Rx.

4.2. CALIBRATION AND VALIDATION PROCEDURES FOR THE TEMPERATURE SENSOR

In the context of the experimental setup for the temperature sensor, calibration assumes a pivotal role in ensuring the accuracy and reliability of temperature measurements [15]. Given the challenges associated with directly controlling temperature in the experimental environment, a distinctive feature of the Pt100 comes into play. Instead of physically manipulating the temperature, the Pt100 can be substituted with a known (precision) resistor.

This approach entails a selection of resistors, each designed to provide a distinct voltage output corresponding to a specific temperature value. By choosing precision resistors in the range of 0 to 140Ω, a comprehensive span of reference temperature points is established, in this case, 0 ºC to 103.9 ºC. The experiment's temperature range is designed

to be used for a range of 0 ºC to 100 ºC. These emulated temperature values corresponding to each resistor's voltage output are calculated in advance providing the user the equivalent of putting the circuit in a controlled environment with a thermometer.

These reference points serve as anchors to build the calibration curve of the circuit, against which all subsequent temperature measurements are adjusted.

Once this groundwork is laid, the calibration process within the application is straightforward. For every resistor used in the simulation, a known temperature value is inputted into the system. Simultaneously, the voltage generated by the circuit is measured. This essential procedure establishes a direct and accurate relationship between voltage and temperature, forming the bedrock for precise temperature measurements throughout the experiment.

With calibration duly completed and the system primed for accurate measurements, the experimental process can progress to data acquisition. This calibrated setup ensures that temperature measurements align with the selected reference temperature points, validating the reliability and validity of the acquired data.

4.3. RESULTS AND ANALYSIS

This segment offers a detailed examination of the obtained data, charting a path through key graphical representations and calibration equations. Additionally, we explore a noteworthy observation, shedding light on the impact of resolution on the collected data.

The Wheatstone bridge was powered with VREF=1V from one of the analog outputs of the DAQ, and the calibration process was conducted as previously described. These are the calculated theoretical values, with *Vd_theo* representing the theoretical value determined by Equation. 3, *T* the temperature calculated with Equation. 1, and *Vd_exp*. being the measured voltage from the circuit.

Rx(Ω)	Vd_theo (V)	T (°C)	Vd_exp. (V)
100	0,0000	0,0000	0
105	0,0122	12,9870	0,012
110	0,0238	25,9740	0,023
121	0,0475	54,5455	0,045
127	0,0595	70,1299	0,057
133	0,0708	85,7143	0,069
140	0,0833	103,8961	0,081

Fig. 4.3.1 Table with Rx, Vd_theo., T and Vd_exp. values powered at 1V With 6001 DAQ

It is noteworthy that *Vd_exp* falls slightly below the anticipated value in comparison to *Vd_theo*. This discrepancy may arise from the fact that, in line with the 6001 specifications, the maximum current it can provide to an external circuit through this analog output channel is limited to 5 mA. However, in the present circuit, the required current is approximately: $\frac{1V}{100\Omega} = 10mA$.

This issue can be solved by either using an external power source for the circuit, like a battery, or by increasing the R1 and R2 values to 1KΩ instead of 100Ω.

R1 and R2 values (Ω)	Vref_exp (V)
100	0,883
1K	1,001

Fig. 4.3.2 Comparison between R1 and R2 values and how Vref is affected

Furthermore, a notable trend emerges from our analysis: the discernible effect of resolution on the acquired data.

In the context of a USB-6001 DAQ system coupled with a Pt100 TF101k temperature sensor operating within a Wheatstone bridge configuration, certain specifications are pertinent.

The resolution, this parameter designates the minimum distinguishable variation in voltage that the DAQ can accurately detect. For the USB-6001 DAQ, with an ADC

resolution of 14 bits, and an input range of ±10 V, the resolution is calculated at 1.22 mV (Equation. 4).

$$Resolution = \frac{10\,V - (-10\text{V})}{2^{14}} = 1.22\text{mV}$$

Equation. 4 Voltage resolution of 6001 DAQ

In addition, bridge sensitivity characterizes the responsiveness of the Wheatstone bridge to alterations in temperature. In this specific instance, it is specified at 0.8 mV per degree Celsius (°C).

With these two things temperature resolution can be calculated. This refers to the smallest change in temperature that a measuring system or instrument can detect or differentiate. It is a measure of the system's ability to distinguish between two temperature values that are very close together.

$$\Delta T = \frac{(ADC\ resolution)}{Bridge\ sensitivity} = \frac{1.22\text{ mV}}{0.8\,\frac{\text{mV}}{\text{°C}}} = 1.5\text{°C}$$

Equation. 5 Temperature resolution for DAQ 6001 with Pt100

This computation yields a temperature resolution of approximately 1.5°C. This means that changes in temperature are quantified in 1.5ºC steps, as it can be observed in (Equation. 5).

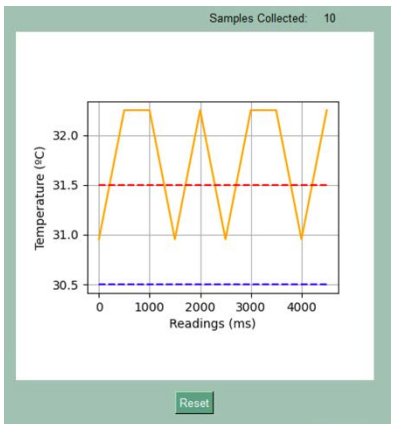

Fig. 4.3.3 Data acquisition plot for DAQ 6001 where temperature resolution can be observed

In DAQs like the 6002, this temperature resolution step is lower as the ADC resolution is higher (16 bits instead of 14) [16].

The data obtained from the experimental setup serves as the cornerstone of our analysis.

4.3.1. CONSIDERATION OF HOW THE PROGRAM CAN FACILITATE STUDENT ENGAGEMENT AND EXPLORATION IN EXPERIMENTAL SETUP

At the core of this project lies the temperature sensor and, by extension, the circuit where it resides.

The application's primary request from the circuit is to provide a voltage signal to read. This means that as long as the student provides an appropriate voltage measurement for the application, there is freedom in the circuit setup. The student can replicate the specific experiment outlined in this project with the Pt100 temperature sensor or they can explore alternative temperature sensors and circuit configurations, fostering a culture of curiosity and experimentation.

In the realm of engineering laboratories, students often grapple with predefined experiments and limited room for experimentation. In contrast, this application empowers students to embark on a journey of discovery. They can easily set up diverse temperature sensors, create intricate circuitry with various components, and observe how these elements interact.

By providing students with the possibility to deviate from the established setup, this program not only enhances their understanding of temperature sensing and data acquisition but also nurtures their innate curiosity and problem-solving skills.

5. CONCLUSIONS

5.1. OVERVIEW OF RESEARCH OBJECTIVES AND MAIN FINDINGS

The primary objective of this project was to create an educational application that could facilitate temperature sensing and data acquisition in order to substitute traditional tools like LabVIEW. The underlying goal was to introduce students to an open-source, Python-based application designed for electronic instrumentation use.

Throughout this project, several key milestones were achieved. These include the successful integration with Python of a flexible and adaptable circuit for temperature measurement, which interfaces with a NI DAQ device, and successfully ensures data acquisition.

One of the accomplishments of this endeavor was the development of a user-friendly application that empowers students to engage in a spectrum of activities. The application allows students to perform essential tasks such as calibration, data acquisition, and real-time data visualization. It provides a comprehensive learning experience, enabling students to apply their classroom knowledge to practical experiments.

As this project progressed, it offered a unique learning opportunity. Valuable insights were gained into the complexities of integrating software and hardware, establishing communication between different components, and planning and building a functional application. All while problem-solving along the way, thus giving a taste of project management and undertaking in the engineering field.

5.2. EXPLORATION OF THE PROGRAM'S POTENTIAL TO ENHANCE LEARNING EXPERIENCES IN TEMPERATURE SENSING AND DATA ACQUISITION

As delved into in previous chapters, this application offers a comprehensive, hands-on learning experience that combines theoretical knowledge and real-world application. It serves as a tool for students seeking to deepen their understanding of data acquisition and interpretation.

By offering a hands-on approach to the concepts of temperature sensing and data acquisition, students can grasp the practical implications of the theories they study in classrooms. This practical acquaintance enhances their comprehension and retention of key concepts, making the learning process more tangible and engaging.

Beyond its role as a data acquisition tool, the program presents an additional benefit to students: an opportunity to explore and understand coding. As already established, Python is rapidly emerging as a standard in various scientific and engineering domains. By the possibility of interacting with the application's code, students gain valuable experience in this coding language. This not only cultivates their computational thinking but also imparts a valuable skill that will accompany them on their journey toward becoming engineers.

5.3. CONCLUDING REMARKS

In this research endeavor, the project successfully achieved its primary research objectives. Delivering an educational application that simplifies temperature sensing and data acquisition. Moreover, the project sets the stage for further exploration and expansion. It provides students, educators, and enthusiasts with a foundation to delve deeper into the realms of temperature sensing and data acquisition. The open-source nature of the application, coupled with the flexibility of the integrated hardware, offers a wealth of possibilities for customization and experimentation. Students and non-students alike can leverage this project as a launchpad for their explorations in the field.

This research journey not only met its objectives but also illuminated the path for future endeavors. It underscores the significance of practical, hands-on experiences in the realm of electronic instrumentation. It reaffirms the value of open-source tools and collaborative efforts in driving innovation and learning. As this chapter concludes, it is recognized that this project is not merely a destination but a steppingstone towards a broader landscape of possibilities in temperature sensing, data acquisition, and beyond.

6. BIBLIOGRAPHY

[1] Instruments, National, "NI," [Online]. Available: https://www.ni.com/es-es/shop/product/labview.html. [Accessed September 2023].

[2] National Instrument, “About NI,” [Online]. Available: https://www.ni.com/en/about-ni.html. [Accessed September 2023].

[3] B. Stroustrup, The C++ programming language, Addison-Wesley Professional, 2013.

[4] H.-P. Halvorsen, "Python for Science and Engineering," 2019. [Online]. Available: https://www.halvorsen.blog/documents/programming/python/python.php.

[5] "NI-DAQmx Python Documentation," [Online]. Available: https://nidaqmx-python.readthedocs.io/en/latest/. [Accessed September 2023].

[6] "PySimpleGUI," [Online]. Available: https://www.pysimplegui.org/en/latest/. [Accessed September 2023].

[7] "USB-6001 Specifications," [Online]. Available: https://www.ni.com/docs/en-US/bundle/usb-6001-specs/resource/374369a.pdf. [Accessed September 2023].

[8] "USB-6001/6002/6003 User Guide," [Online]. Available: https://www.ni.com/docs/en-US/bundle/usb-6000-6001-6002-6003-features/resource/374259a.pdf. [Accessed September 2023].

[9] "tkinter — Python interface to Tcl/Tk," [Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed September 2023].

[10] "NI-DAQ™mx Driver," [Online]. Available: https://www.ni.com/en/support/downloads/drivers/download.ni-daq-mx.html#484356. [Accessed September 2023].

[11] "NumPy documentation," [Online]. Available: https://numpy.org/doc/stable/index.html. [Accessed September 2023].

[12] "SciPy," [Online]. Available: https://scipy.org/. [Accessed September 2023].

[13] "Matplotlib — visualization with python," [Online]. Available: https://matplotlib.org/. [Accessed September 2023].

[14] R. P. Areny, Sensores y acondicionadores de señal, Sevilla Marcombo Boixareu, 2003.

[15] M. Á. Pérez García, Instrumentación electrónica, Madrid: Thomson-Paraninfo, 2011.

[16] National Instruments, "ni.com," [Online]. Available: https://www.ni.com/docs/en-US/bundle/usb-6002-specs/resource/374371a.pdf. [Accessed September 2023].

[17] Texas Instruments, "LM35 Precision Centigrade Temperature Sensors," [Online]. Available: https://www.ti.com/lit/ds/symlink/lm35.pdf. [Accessed September 2023].

APPENDIX A: ASSETS AND ATTRIBUTION

The following icons used in this project were sourced from Flaticon, a platform that provides a wide range of icons for various purposes. Each icon has been attributed to its respective creator.

Fig.1 Original images from Flaticon

Snack free icon: Nature icons created by max.icons – Flaticon (https://www.flaticon.com/free-icons/nature)

Thermometer free icon: Thermometer icons created by Freepik – Flaticon (https://www.flaticon.com/free-icons/thermometer)

Led free icon : Led icons created by Smashicons – Flaticon (https://www.flaticon.com/free-icons/led)

Switch free icon : Toggle button icons created by Creatype – Flaticon (https://www.flaticon.com/free-icons/toggle-button)

APPENDIX B: STUDENT'S GUIDE

This resource was created to provides comprehensive insights into temperature sensing and data acquisition for students when using PyroDAQ. A PDF version can be downloaded from the project's GitHub repository found in "APPENDIX C: CODE" section.

PYRODAQ STUDENT'S GUIDE:

EXPLORING TEMPERATURE SENSING WITH PYTHON AND CONTROLLING NATIONAL INSTRUMENTS DAQ

INTRODUCTION

Welcome to PyroDAQ – your gateway to python driven temperature sensing. This guide is your key to mastering temperature measurement using our application and National Instrument's equipment.

Temperature sensing holds pivotal importance in various fields, from electronics to industry. PyroDAQ is born out of the need to merge Python programming capabilities and accessible, comprehensible code, with National Instrument's precision, enabling you to navigate temperature measurement confidently.

In this guide we'll lead you through installation, circuit setup, and program functionality. Get ready to explore temperature sensing through PyroDAQ – where hardware and software unite for precise measurements. Let's dive in!

ESSENTIAL CONCEPTS IN TEMPERATURE SENSING

In this section, we'll delve into the essential building blocks that underpin accurate temperature measurements and equip you to navigate electronic instrumentation.

Sensitivity:

Sensitivity quantifies a sensor's responsiveness to temperature fluctuations. A highly sensitive sensor detects even subtle temperature changes, enabling accurate and detailed measurements.

Resolution:

Temperature resolution signifies the smallest temperature change a sensor can detect. A higher resolution allows finer distinctions, crucial for precision in temperature-sensitive applications.

The resolution of an ADC refers to the smallest increment of analog input voltage that can be accurately represented as a discrete digital value.

For USB-6001 DAQ:

ADC Resolution: 14 bits
ADC FS voltage: ±10 V

Resolution = (ADC FS voltage) / (2^ADC Resolution)
Resolution = (10 V - (-10 V)) / (2^14) = 1.22 mV

For USB-6002 DAQ:

ADC Resolution: 16 bits
ADC FS voltage: ±10 V

Resolution = (10 V - (-10 V)) / (2^16) = 305 µV

For USB-6211 DAQ:

ADC Resolution: 16 bits
ADC FS voltage: ±10 V, ±5 V, ±1 V, ±0.2 V (with 5% overrange)

Resolution = 320, 160, 32, 6.4 µV

To estimate the resolution in temperature measurement for a data acquisition system using a temperature sensor with a voltage output, it is essential to know the sensitivity of the sensor. The sensitivity of the temperature sensor refers to how much the sensor's output

voltage changes in response to a unit change in temperature. The higher the sensor's sensitivity, the higher the temperature measurement resolution you can achieve with the data acquisition system. Resolution is typically calculated as the ratio between the minimum detectable variation in the sensor's output voltage and the sensor's sensitivity. The lower this ratio, the higher the temperature measurement resolution.

For USB-6001 DAQ with LM35 sensor and:

ADC Resolution: 1.22 mV
LM35 sensitivity: 10 mV/°C

Temperature resolution = (ADC Resolution) / (LM35 sensitivity)

Temperature resolution = 1.22 mV/10 mV/°C = 0.122 °C.

For USB-6001 DAQ with a Pt100 TF101k in a Wheatstone bridge with 0.8 mV/°C sensitivity:

ADC Resolution: 1.22 mV
Bridge sensitivity: 0.8 mV/°C

Temperature resolution = (ADC Resolution) / (Bridge sensitivity)

Temperature resolution = 1.22 mV/ 0.8 mV°C =1.5 °C.

Offset:

Offset accounts for inherent sensor errors by adding or subtracting a constant value from the output. Managing offsets fine-tunes accuracy, especially in low-temperature ranges.

Sampling Rate

Sampling rate dictates how often measurements are taken. A suitable sampling rate captures rapid temperature changes without missing vital data.

Calibration

Calibration stands as a critical process that ensures your temperature sensor's readings remain accurate and reliable. It involves adjusting the sensor's output to match a known reference value, thereby correcting any inherent biases or deviations that may arise over time.

Linear Calibration

For sensors with a linear, or relatively so, relationship between their output and the measured quantity (PT100, LM35, limited range Thermocouples, etc.), linear calibration is often employed.

Non-linear Calibration

Not all sensors exhibit linear behavior. Some sensors, especially those with intricate response curves, demand non-linear calibration (thermistor, RTD, thermocouples, etc.). In these cases, more sophisticated equations are employed. Non-linear calibration handles the intricacies of the sensor's behavior and ensures accurate compensation across its entire operating range.

Choosing the Right Calibration Approach

The choice between linear and non-linear calibration hinges on the sensor's characteristics and the desired accuracy. Linear calibration is simple and effective when the sensor's deviation from linearity is minor. Non-linear calibration, while more complex, accommodates sensors with non-linear behaviors and offers better accuracy across a wider range of conditions.

5

Data Acquisition Device (DAQ)

A component designed to capture, measure, and analyze real-world data from various physical phenomena. It serves as a bridge between the analog world and digital processing, enabling precise data collection for analysis and control.

Wheatstone Bridge

A Wheatstone Bridge is a fundamental circuit used in electronics to measure resistances and detect changes in resistance with high precision. It consists of four resistors arranged in a diamond shape, with a power source connected across one diagonal. When the bridge is balanced, the ratio of resistances is equal on both sides. This setup allows you to measure an unknown resistance by adjusting known resistances until the bridge is balanced.

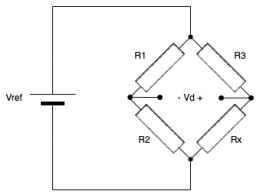

Figure 1. Wheatstone Bridge Diagram

In the following sections, we'll dive deeper into the intricacies of PyroDAQ and its integration with National Instrument's equipment. Get ready to translate theory into practical mastery!

SYSTEM REQUIREMENTS

Before embarking on your temperature sensing journey, let's ensure you have everything you need. Here's a breakdown of the hardware and software necessities:

Hardware Requirements

1. **National Instruments DAQ (USB-6001, 6002, or 6211):** Think of it as the bridge between the real world and your computer. It will also power your circuit (external power source can also be used).

2. **Small flat screwdriver:** necessary for connecting wires to the DAQ.
3. **Circuit Setup:** breadboard, It provides a convenient platform for assembling and connecting the temperature sensor, precision resistors, and wires.
4. **Temperature Sensor:** You can choose a PT100 or LM35 sensor, among others.
5. **Precision Resistors:** These help fine-tune your measurements. They'll be valuable when calibrating.
6. **Wires:** You'll need a few in different colors such as red, black, blue, ...

Software Requirements

1. **Operating System:** Windows will do the job.
2. **PyroDAQ App:** This is your window into the temperature sensing world, it helps you see and understand what's happening with your setup.

Checklist:

- ☐ National Instruments DAQ (USB-6001, 6002, or 6211)
- ☐ Screwdriver
- ☐ Breadboard
- ☐ Temperature Sensor (PT100, LM35, etc.)
- ☐ Precision Resistors (100, 105, 110, 121, 127, 133 and 140 Ω)
- ☐ Wires
- ☐ Computer with Windows Operating System
- ☐ PyroDAQ App (Download and installation process explained next).

With these essentials in place, you're all geared up to explore temperature sensing and dive into the world of electronic instrumentation using PyroDAQ.

INSTALLATION

To get started with PyroDAQ, you'll need to follow a few simple steps to install the application on your computer. Here's a detailed guide:

1. Access the GitHub Repository

First, you'll need to visit the PyroDAQ GitHub repository. You can find the link to the repository here. This is where you'll find all the necessary files to install PyroDAQ.

2. Read the README.md

Once you're on the GitHub page, navigate to the README.md file. This file contains comprehensive instructions on how to install and set up PyroDAQ on your computer. It's your go-to resource for the installation process.

3. Follow the Installation Steps

The README.md file will provide step-by-step instructions for installing PyroDAQ. These steps typically include downloading the necessary files, installing any dependencies, and configuring your environment.

By following these steps and carefully reading the instructions in the README.md file on the GitHub repository, you'll be able to successfully install PyroDAQ on your computer. Happy installation!

GETTING STARTED

Let's kick off your journey by diving into the initial setup process. We will be exploring the Wheatstone bridge circuit setup with a Pt100, but feel free to diverge into other circuit examples.

As you embark on this adventure, it's essential to start with the right foot forward. Ensure you have all the hardware and software requirements in place, as outlined in the "System Requirements" section. This includes your National Instruments DAQ, circuit components, temperature sensor, wires, and your computer with the PyroDAQ app installed.

Wheatstone Bridge Setup with a Pt100

In a Wheatstone bridge setup, calibration is the process of establishing a relationship between the electrical signals produced by the bridge and the actual physical quantity you're trying to measure, such as temperature. Precision resistors play a key role in this calibration process, allowing you to simulate different temperature conditions without changing the actual temperature of the environment.

Here's a step-by-step guide:

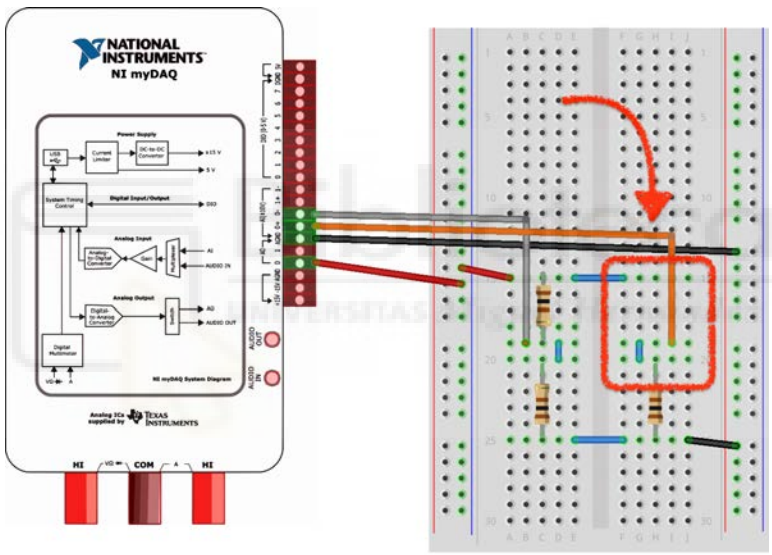

Figure 2. Wheatstone Bridge Circuit Setup

1. Arrange on a breadboard the circuit from Figure 2.

9

a. The three 100-ohm resistors will form the arms of the bridge. Connect them in such a way that one resistor connects to the positive rail, another to the negative rail, and the third resistor connects to the output pin of the bridge.
b. Connect the first of the precision resistors in the remaining place of the wheatstone bridge (where the red box is). Once calibrated, this will be where we connect the Pt100.
c. Referencing Figure 1, connect the breadboard to the DAQ with wires:

Diagram	DAQ
Vin+, Vin -	AO0*, AOGND
Vout+, Vout -	AI0+, AI0-

**AO0 powers the circuit from the DAQ, if desired this can be replaced by another power supply of 1V. If this is changed, keep in mind that the reading configuration may need to be changed from DIFF to something appropriate like RSE.*

Temperature-Resistance table

To do so, fill in the table, this will correlate the known resistance values of the precision resistors with their corresponding temperatures. This table serves as a reference for simulating different temperature points and we will use it in the calibration step later on.

Rt(Ω)	Theoretical Vd (V)	Temperature (°C)	Measured Vd (V)
100			
105			
110			
121			
127			
133			
140			

Formulated spreadsheet: Student Guide Table.xlsx

10

Temperature

The relationship between Pt100 and temperature can be represented by:

$R_t = R_o + \alpha(t - t_o)$, with $R_o = 100\,\Omega$, $t_o = 0ºC$ and $\alpha = 0.00385\,ºC^{-1}$

$R_t = 100 + 0.00385(t - 0) \rightarrow t = ((\frac{R_t}{100} - 1)/0.00385) + 0$

Voltage

For a balanced bridge like this one: $V_{OUT} = V_{IN}(\frac{R_t}{R_t+R_3} - \frac{R_1}{R_1+R_2})$

NAVIGATING PYRODAQ STEP BY STEP

Get ready to explore the ins and outs of this powerful tool as we walk you through each step of calibrating and acquiring data from a Wheatstone bridge setup.

LAUNCH

1. The first window you'll encounter will kindly ask you to select the specific model of your DAQ. This information can be found on the underside of your DAQ.

   ❗ Make sure your DAQ is called "Dev1" as the app won't recognize it if it's not. This can be checked and changed with the NI MAX software, in devices and interfaces.

CALIBRATION

In this initial stage, we're going to calibrate your temperature sensing setup using precision resistors. Calibration is like teaching your system how to speak the language of temperature changes. Here's how it works:

2. The Calibration Menu appears, ready to log your calibration settings. Once a calibration is set, it can be later used as they're saved.

   Two options appear: Temperature and Voltage Relation and Input Direct Expression.

For our Wheatstone bridge setup, let's opt for the Temperature and Voltage Relation. This is because we're simulating different temperatures using precision resistors. For known calibrations like for a LM35 circuit setup, the direct input expression would be useful.

3. Click "Calibrate" to proceed.

Temperature and Voltage Relation

4. In this window, you'll first need to select an expression type. We'll go with the linear equation, least squares method for this tour, but feel free to explore other methods.
5. Under input data, toggle "Measure". If you'd instead like to input the calculated voltage toggle for "Type In".
6. Input the corresponding temperature value from your table and either click "Enter" or press the Enter key.
7. Do so for each precision resistor, changing the resistor in the circuit as you go. You can check if your measured values correspond with your previously calculated ones.

   It's normal for the real values to differ a bit from the calculated values.

8. If you make any mistakes, don't worry. You can delete the last input by clicking "delete", or all of them by clicking "Clear".

As you add data points, you'll start to see your calibration plot taking shape.

9. Once all points are added, click "Choose" to finalize the calibration.

Back in the Calibration Menu, you can keep logging calibrations, but for now, let's move forward.

10. Click "Acquire Data" to proceed.

DATA ACQUISITION AND VISUALIZATION

Now, we're ready to use the calibration expression to acquire actual data from the Pt100 sensor:

11. Connect your Pt100 temperature sensor in place of the precision resistors.

12

12. In the appearing window, you'll see your chosen calibration at the top. If you need to change it, click "recalibrate."
13. Explore control features like temperature alarms. Set minimum and/or maximum alarms by entering desired values and clicking on "Set" (e.g. 25ºC, 28º) . Dashed lines will appear in the graph with their corresponding colors.

    To delete the alarms, click on "disable".

Choose your data acquisition type:

14. For on-demand acquisition:
    a. Select the option and click "acquire data."
    b. Adjust the time interval with the slider. For more precision, click on either side of the slider.
    c. Click on "Stop" when you're ready.
15. For finite sampling:
    a. Select the option.
    b. Input the number of samples and time interval (e.g. 10, 2).
    c. Click "Acquire Data."
    d. Once finished, if desired, save the data as a CSV file by clicking on "Save Data".

       Input a name for the file and open it on your computer to review parameters, alarms, and acquired values.

       This CSV file can easily be opened in other platforms such as excel to further analyze the data.

With this tour, you've unlocked the basics of PyroDAQ. Repeat and experiment as much as you like, delving into different calibrations and sensors. Let your curiosity lead the way as you master the art of temperature sensing!

APPENDIX C: CODE

What follows is the complete source code utilized in developing the PyroDAQ application.

This codebase is available for reference and further exploration at the project's GitHub repository, which can be accessed via the following link: GitHub Repository (https://github.com/danllaq/PyroDAQ.git).

PyroDAQ.bat

```
@echo off
title PyroDAQ

rem Activate the virtual environment
call PyroDAQvenv\Scripts\activate

rem Run the PyroDAQ application
python main.py
```

README.md

PyroDAQ Ø=Ü Ø<ß!þ
PyroDAQ is a Python application with a graphical user interface (GUI) designed for interacting with
National Instruments DAQ (Data Acquisition) devices for temperature sensing. Ø=Ý%

Prerequisites Ø=ÜË

Before you begin, make sure you have Python and Pip installed on your system and that you're using Windows. Ø=Ü»

Python Installation Ø=Ü
1. Visit the [Python Dowloads](https://www.python.org/downloads/) page.
2. Download the installer for the version *3.11.0*.
3. Run the installer and follow the installation instructions.
4. During the installation, make sure to check the box that says "Add Python to PATH".
5. After installation, open a command prompt and check that Python is installed by running:
```batch
python -- version
```

Pip Installation Ø=Üæ

When installing Python, pip should also be included. To check if it's installed run:
```batch
pip --version
```
If for any reason there's a problem refer to [pip documentation](https://pip.pypa.io/en/stable/installation/).

Driver Installation &™þ

To use PyroDAQ, you need to install the NI-DAQmx (version 2023 Q1).

Downloading and installing the NI-DAQmx driver is essential because it provides the necessary software components for your
computer to communicate with and control National Instruments DAQ hardware devices that this project uses.

Follow these steps to download and install the driver:
1. **Download the NI-DAQmx Driver:**
   - Visit the official NI-DAQmx driver download pager: [NI-DAQmx Driver Download](https://www.ni.com/es/support/downloads/drivers/download.ni-daq-mx.html#477807).
   - Choose Windows OS and the 2023 Q1 version.
   - Click download button and save to computer.
2. **Installation:**
   - Locate the .exe file and double-click.
   - Follow the instructions.
3. **Verification:**
   - Verify the installation by opening NI MAX.
   - Open the "My System > Software".
   - You should see the driver and the correct version.

Setting up the project Ø=Þàþ

Downloading the project Ø=Üå
1. From [PyroDAQ GitHub page](https://github.com/danllaq/PyroDAQ) download zip file.
2. Extract the zip file to your prefered directory.

Creating a virtual environment Ø<ß
In order to isolate dependencies for this project, we're going to create a virtual environment. It's important to note that
the project and dependencies are going to be inside the venv but *Python and pip **should not be** in the venv* 'W

To quickly set up and configure this project, follow these steps:
1. Copy the path of your project's directory.
1. Open a command prompt and open the directory with the following command (substituting for your actual path):
```bash
cd C:\Users\<user_name>\PyroDAQ-main
```
3. Run the setup script:
```bash
setup.bat
```
4. Wait until the setup has finished, this will be indicated with ```Setup completed!``` it might take a few moments.

Running the Program %¶þ

After you have completed the prerequisites and set up the project, follow these steps to run PyroDAQ:
1. **Launch PyroDAQ:**
   You have some options for running the program:
   - In a command line, from your project directory, run:
     ```bash
     PyarDAQ.bat
     ```
   - You can also double click on the file `PyroDAQ.bat`
   - Or in a command line your project directory, run:
     ```bash
     PyroDAQvenv\Scripts\activate
     python main.py
     ```
3. **Interact with the program**
    - Once the program is running, the GUI for PyroDAQ should appear
    - You can now connect you DAQ and use the GUI to interact with it for temperature sensing and other data tasks
4. **Student's Guide**
   - You can find more instructions and a guide through the program in the attached pdf "Student's Guide"

That's It! You're Set to Blaze a Trail with PyroDAQ Ø=Ü Ø<ß!þ

Congratulations! You've successfully set up PyroDAQ and are now ready to embark on your data acquisition adventures. Whether you're a seasoned engineer, a curious hobbyist, or somewhere in between, we hope PyroDAQ adds some heat to your temperature sensing projects!

Remember, the world of data acquisition is vast and filled with exciting challenges. So, go forth, measure temperatures, and conquer your data like a pro.

Happy data collecting, and stay toasty! Ø=Ý%Ø=Þ

main.py

```python
import sys
import os

import src.app.appDAQ as daq
import src.app.appCalibrationMethod as calibration_method
import src.app.appDataAcquisition as data_acquisition

# Gets the path of the current script
current_dir = os.path.dirname(os.path.abspath(__file__))

# Adds the parent directory (project root) to the Python path

parent_dir = os.path.dirname(current_dir)
sys.path.append(parent_dir)

def main():

    # --- DAQ SELECTION ---
    niDAQ = daq.run_select_daq()
    while not niDAQ.is_exit_requested():
        # --- CALIBRATION ---
        calibration_method.run_calibrate(niDAQ)
        if niDAQ.is_exit_requested():
            niDAQ.exit()
            continue
        # --- ACQUIRE DATA ---
        data_acquisition.run_data_acquisition(niDAQ)

if __name__ == "__main__":
    main()
```

requirements.txt

```
nidaqmx==0.6.5
matplotlib==3.7.1
PySimpleGUI==4.60.5
numpy==1.24.2
scipy==1.11.1
```

setup.bat

```
@echo off
title PyroDAQ setup

rem Creates virtual environment
echo Creating the virtual environment...
python -m venv PyroDAQvenv

rem Activates the virtual environment
echo Activating the virtual environment...
call PyroDAQvenv\Scripts\activate

rem Installs dependencies
echo Installing the required dependencies...
pip install -r requirements.txt

rem Deactivate the virtual environment
echo Deactivating the virtual environment...
call deactivate

rem Display message to confirm that the script has completed

echo Setup completed!
```

src/app/appCalibrationMethod.py

```
import src.gui.guiCalibrationMethod as guiCalibrationMethod
import src.app.appTempVoltCalibrate as appTempVoltCalibrate
import src.app.appExpressionInputCalibrate as appExpressionInputCalibrate

def run_calibrate(niDAQ):
    """
    Runs calibrate menu
    :param niDAQ: object with DAQ information and where all things related is
stored
    :return:
    """
    while True:
        try:
            if niDAQ.is_calibration_set():
                layout =
guiCalibrationMethod.layout_with_expression(niDAQ.calibrations_log)
                # launches window where calibration expression and/or method
will be selected
                window =
guiCalibrationMethod.calibration_method_window(layout)
                method =
guiCalibrationMethod.run_calibration_method_window(window, niDAQ)
            else:
                layout = guiCalibrationMethod.get_layout_no_calibration()
                window =
guiCalibrationMethod.calibration_method_window(layout)
                method =
guiCalibrationMethod.run_calibration_method_no_calibration_window(window)
            match method:
                case 'TEMP_VOLTAGE':
                    calibration =
appTempVoltCalibrate.run_temp_volt_calibrate(niDAQ)
                    if calibration is not None:
                        niDAQ.add_calibration_to_log(calibration)
                        niDAQ.set_calibration(repr(calibration))
                case 'EXPRESSION_INPUT':
                    calibration =
appExpressionInputCalibrate.run_expression_input_calibrate(niDAQ)
                    if calibration is not None:
                        niDAQ.add_calibration_to_log(calibration)
                        niDAQ.set_calibration(repr(calibration))
                case 'ACQUIRE_DATA':
                    if niDAQ.is_calibration_set:
                        break
                    else:
                        raise ValueError("No calibration assigned")
                case 'EXIT':
                    niDAQ.set_exit_request()
                    break
                case _:
                    raise ValueError("Wrong calibration method chosen.")
        except ValueError as e:
            print(f"Error: {e}")
```

src/app/appDAQ.py

```python
import src.daqTools as dt
import src.gui.guiDAQ as guiDAQ

def run_select_daq():
    """
    Runs daq selection
    :return:
    """
    while True:
        try:
            # creates object where DAQ information is stored

            modelsDAQ, exitFlag = guiDAQ.select_daq_window(dt.modelsDAQ)
            # DAQ initiation with its corresponding model

            niDAQ = dt.niDAQ(modelsDAQ, exitFlag)
            if not exitFlag:
                niDAQ.initiate_daq()
            return niDAQ
        except ValueError as e:
            guiDAQ.no_daq_detected_popup(e)
```

src/app/appDataAcquisition.py

```
import src.gui.guiDataAcquisition as guiDataAcquisition

def run_data_acquisition(niDAQ):
    """
    Runs data acquisition
    :param niDAQ: object where data will be stored
    :return:
    """
    # launches window where the user can input calibration expression

    window, fig, figure_canvas_agg =
guiDataAcquisition.data_acquisition_window(niDAQ.calibration)
    niDAQ.set_task_start(1)
    niDAQ.set_task_write(1)
    guiDataAcquisition.data_acquisition_window_behavior(niDAQ, window, fig,
figure_canvas_agg)
    niDAQ.set_task_stop(1)
```

src/app/appExpressionInputCalibrate.py

```
import src.gui.guiExpressionInputCalibrate as guiExpressionInputCalibrate

def run_expression_input_calibrate(niDAQ):
    """
    Runs expression input calibration
    :param niDAQ: object where calibration will be stored
    :return: object with the calibration information
    """
    # launches window where the user can input calibration expression

    window, fig, figure_canvas_agg =
guiExpressionInputCalibrate.expression_calibrate_window()
    # the window returns either a '1' if the user has chosen an expression or
-1 if they want to go back or close window

    niDAQ.set_task_start(1)
    niDAQ.set_task_write(1)
    calibration =
guiExpressionInputCalibrate.expression_input_calibrate_window_behavior(niDAQ,
window,

           fig, figure_canvas_agg)
    niDAQ.set_task_stop(1)
    return calibration
```

src/app/appTempVoltCalibrate.py

```
import src.gui.guiTempVoltCalibrate as guiTempVoltCalibrate

def run_temp_volt_calibrate(niDAQ):
    """
    Runs temperature-voltage relation calibration
    :param niDAQ: object where calibration will be stored
    :return: object with the calibration information
    """
    # launches window where the user can input voltage and temperature and an
expression will be calculated
    window, fig, figure_canvas_agg =
guiTempVoltCalibrate.temp_volt_calibrate_window()
    niDAQ.set_task_start(1)
    niDAQ.set_task_write(1)
    # the window returns either a '1' if the user has chosen an expression or
-1 if they want to go back or close window

    calibration =
guiTempVoltCalibrate.temp_volt_calibrate_window_behavior(niDAQ, window, fig,
figure_canvas_agg)
    niDAQ.set_task_stop(1)
    return calibration
```

src/calibrationTools.py

```python
from abc import ABC, abstractmethod
import src.guiTools as gt
import numpy as np
from scipy.optimize import curve_fit
import warnings

def linear_func(x, m, n):
    """
    Linear function method
    :param x: variable
    :param m: slope
    :param n: y interception
    :return: linear function
    """
    return m * x + n

def non_linear_func(x, a, b, c):
    """
    Non-linear function method
    :param x: variable
    :param a: grade 2 param
    :param b: grade 1 param
    :param c: constant
    :return: non-linear function
    """
    return a * x ** 2 + b * x + c

class Calibration(ABC):
    """
    Calibration parent class, everything related to the calibration when it's
being set goes here. Once
    It's set, it goes to the niDAQ class object
    """

    def __init__(self, expression_type):
        """
        Initiates calibration object given the expression type
        :param expression_type: string, types: 'LINEAR_EQUATION',
'NON_LINEAR_EQUATION'
        """
        self.expression_type = expression_type  # types: 'LINEAR_EQUATION',
'NON_LINEAR_EQUATION'
        self.parameters = {}  # dictionary where parameters are stored
{'coefficient_g2', 'coefficient_g1', 'constant'}

        self.data = []
        self.interpolation_points = []  # only used for interpolation


    def __len__(self):
        """
        When len(object) is used, returns length of data list
        :return: number of points in data list
        """
        return len(self.data)

    def __getitem__(self, index):
        """
```

```
        When object[index] is used, accesses data list to retrieve value
        :param index: index to access
        :return: pair with [voltage, temperature]
        """
        return self.data[index]

    def __setitem__(self, index, voltage_temperature: list):
        """
        When object[index] is used, accesses data list to assign value
        :param index: index to access
        :return:
        """
        check_all_floats(voltage_temperature[0], voltage_temperature[1])
        self.data[index] = voltage_temperature

    def __delitem__(self, index):
        """
        When using del object[index], deletes item in data list
        :param index: index to access
        :return:
        """
        del self.data[index]

    def set_chosen_points(self, chosen_points: list):
        """
        Assign the chosen points pair selected by the user
        :param chosen_points: two pairs in a list
        :return:
        """
        self.interpolation_points = chosen_points

    def set_data_list(self, data):
        """
        Takes a list (data) containing pairs of data points and assigns it to
a list in an object.
        :param data: list of data pair points
        :return:
        """
        self.data = data

    def get_parameter(self, parameter_name):
        """
        Accesses equation parameter dictionary and returns desired value
        :param parameter_name: name of the equation parameter
['coefficient_g2', 'coefficient_g1', 'constant']
        :return: equation parameter value
        """
        return self.parameters.get(parameter_name)

    def get_data(self):
        """
        Returns the data points stored by the user
        :return: list of pairs
        """
        return self.data

    def update_parameters(self, parameters_dict):
        """
        Takes a dictionary (parameters_dict) containing the parameter names
and their corresponding values.
        The update method of the parameters dictionary is then used to update
```

```
the parameter values in another object.
        :param parameters_dict: dictionary where equation parameter values
are stored
        :return:
        """
        self.parameters.update(parameters_dict)

    def add_voltage(self, voltage):
        """
        Given a voltage value, will add to the data list
        :param voltage: Voltage value, float
        :return:
        """
        [voltage] = gt.to_number_n_dec(gt.N_DECIMALS, voltage)
        temperature = self.calculate_temperature(voltage)
        self.data.append([voltage, temperature])

    def add_data(self, voltage_temperature: list):
        """
        Given a voltage and temperature pair, adds to de data
        :param voltage_temperature: voltage and temperature pair
        :return:
        """
        self.data.append(voltage_temperature)

    def clear_data(self):
        """
        Clears table list
        :return:
        """
        self.data.clear()

    def update_data(self):
        """
        Updates data points
        :return:
        """
        for i, (voltage, temperature) in enumerate(self.data):
            new_temperature = self.calculate_temperature(voltage)
            self[i] = [voltage, new_temperature]

    def sort_x(self):
        """
        Sorts data points by x
        :return: list of x values
        """
        return gt.get_sorted_nth_elements(self.data, n=0)

    def sort_y(self):
        """
        Sorts data points by x
        :return: list of y values
        """
        return gt.get_sorted_nth_elements(self.data, n=1)

    def is_linear(self):
        """
        Checks if object is linear
        :return: boolean, true if it is
        """
        return self.expression_type == 'LINEAR_EQUATION'
```

```
    def is_nonlinear(self):
        """
        Checks if object is non-linear
        :return: boolean, True if it is
        """
        return self.expression_type == 'NON_LINEAR_EQUATION'


    def is_type(self, expression_type):
        """
        Checks if the object has the same expression type
        :param expression_type: LINEAR_EQUATION or NON_LINEAR_EQUATION
        :return: True if they are, false if it isn't
        """
        return self.expression_type == expression_type

    def data_exists(self, data_point):
        """
        Checks if voltage is already stored in data
        :param data_point:pair [voltage, temperature]
        :return: True if voltage is already stored
        """
        return any(voltage == data_point[0] for voltage, temperature in
self.data)

    def has_enough_points(self):
        """
        Checks if there are at least 2 points to calculate a linear
expression, or at least 3 for a nonlinear
        :return:
        """
        return (len(self) > 1 and self.is_linear()) or (len(self) > 2 and
self.is_nonlinear())

    def to_linear_calibration(self, m=0, n=0):
        """
        Converts a NonLinearCalibration object to a LinearCalibration one,
updating parameters and passing data
        :param n:
        :param m:
        :return: linear_cal, LinearCalibration object with relevant
information
        """
        if self.expression_type == "NON_LINEAR_EQUATION":
            # creates LinearCalibration object

            linear_cal = LinearCalibration()
            # converts to floats with 3 decimal points

            m, n = gt.to_number_n_dec(gt.N_DECIMALS, m, n)
            # assigns parameters to new object

            linear_cal.update_parameters(parameters_dictionary(m, n))
            # assigns data list to new object
            linear_cal.set_data_list(self.data)
            return linear_cal
        else:
            raise ValueError("Cannot convert to LinearCalibration. Current
equation type is linear.")
```

```
    def to_nonlinear_calibration(self, a=0, b=0, c=0):
        """
        Converts a LinearCalibration object to a NonLinearCalibration one,
updating parameters and passing data
        :param c:
        :param b:
        :param a:
        :return: non_linear_cal, NonLinearCalibration object with relevant
information
        """
        if self.expression_type == "LINEAR_EQUATION":
            # creates NonLinearCalibration object

            non_linear_cal = NonLinearCalibration()
            # converts to floats with 3 decimal points

            a, b, c = gt.to_number_n_dec(gt.N_DECIMALS, a, b, c)
            # assigns parameters to new object

            non_linear_cal.update_parameters(parameters_dictionary(a, b, c))
            # assigns data list to new object
            non_linear_cal.set_data_list(self.data)
            return non_linear_cal
        else:
            raise ValueError("Cannot convert to NonLinearCalibration. Current
equation type is not linear.")

    def draw_expression(self, axes, known_expression):
        """
        Draws expression plot
        :param axes: axes where the figure is
        :param known_expression: calibration expression
        :return:
        """
        if known_expression:
            # Generate points for the plot
            if len(self) > 0:
                x_plot = np.linspace(self.sort_x()[0], self.sort_x()[-1], 100)
            else:
                x_plot = np.linspace(0, 10, 100)

            self.plot_expression(axes, known_expression, x_plot)
        else:
            if self.has_enough_points():
                self.plot_expression(axes, known_expression)

    def update_figure(self, fig, figure_canvas_agg, known_expression,
is_point_selected=False, x_sel_point=None,
                      y_sel_point=None):
        """
        Updates and draws the plot
        :param fig: calibration plot
        :param figure_canvas_agg: canvas for calibration plot
        :param known_expression: boolean, indicates if the figure is being
drawn with a known expression input by user
        :param is_point_selected: boolean if user has selected a point from
table
        :param x_sel_point: x for selected point
        :param y_sel_point: y for selected point
        :return:
        """
```

```
        axes, x, y = gt.get_axes_for_points(fig, self.data)
        self.draw_expression(axes, known_expression)
        gt.draw_points(axes, x, y, 'bo', "Data Points")
        if is_point_selected:
            gt.draw_points(axes, x_sel_point, y_sel_point, 'ro')
        axes[0].legend()
        gt.pack_canvas(figure_canvas_agg)

    def change_in_data(self, win, fig, figure_canvas_agg, known_expression):
        """
        Updates window when there is a change in the data
        :param win: pysimplegui window
        :param fig: calibration plot
        :param figure_canvas_agg: canvas for the calibration plot
        :param known_expression: calibration expression
        :return:
        """
        win['-TABLE-'].update(values=self.data)
        if not known_expression:
            win['-N_SAMPLES-'].update(len(self))
        # if a point was deleted that was used for interpolation, the
interpolation data clears
        if self.is_interpolation_points_in_data():
            self.interpolation_points.clear()
        self.update_figure(fig, figure_canvas_agg, known_expression)

    @abstractmethod
    def calculate_temperature(self, voltage: float):
        """
        Abstract method, given a voltage value it will be overriden by the
appropriate subclass method that
        will calculate the temperature
        :param voltage: Voltage value, float
        :return:
        """
        pass

    def plot_expression(self, axes, known_expression, x_plot=None):
        """
        Abstract method, given an x_plot and axes it will be overriden by the
appropriate subclass method that will
        plot the expression
        :param known_expression:
        :param x_plot: list of x values for the plot to reference
        :param axes: axes where the graph will be plotted on
        :return:
        """
        pass

    def is_interpolation_points_in_data(self):
        """
        Checks if the set interpolation points are in the data
        :return: boolean, True if they are
        """
        return not all(pair in self.data for pair in
self.interpolation_points)

def check_all_floats(*args):
    """
    Given a list of values, checks if they all are float. If any isn't raises
an error.
```

```
    Used when values are inputted by user.
    :param args: equation parameters
    :return:
    """
    for arg in args:
        if not isinstance(arg, float):
            raise ValueError(f"Invalid value: '{arg}' is not float. Type: {
type(arg).__name__}")

def get_sign(number):
    """
    Return '+' if number is positive
    :param number:
    :return:
    """
    return ' ' if number < 0 else ' + '


def parameters_dictionary(*args):
    """
    Given the expression parameters, creates a dictionary format in order to
save to the object
    :param args: calibration parameters, 2 for linear, 3 for non-linear
    :return: dictionary with the format set by the programmer
    """
    if len(args) == 3:
        return {'coefficient_g2': args[0], 'coefficient_g1': args[1],
'constant': args[2]}
    elif len(args) == 2:
        return {'coefficient_g2': None, 'coefficient_g1': args[0], 'constant'
: args[1]}
    else:
        raise ValueError(f"There must be 2 or 3 parameters, {len(args)} were
provided")

class LinearCalibration(Calibration):
    """
    Child class of calibration for the linear expression, here everything
related to linear calibration
    is managed and can convert to non-linear class if needed
    """
    def __init__(self, calculation_method=""):
        """
        Creates a child class of Calibration that if assigned has a type of
calculation method
        :param calculation_method: methods: ['LEAST_SQUARES',
'LINEAR_INTERPOLATION']
        """
        super().__init__("LINEAR_EQUATION")
        self.calculation_method = calculation_method

    def __repr__(self):
        """
        String representation method
        :return: calibration equation in string form
        """
        return f"y = {self.get_parameter('coefficient_g1'):.3f}x
{get_sign(self.get_parameter('constant'))}" \
               f"{self.get_parameter('constant'):.3f}"

    def update_method(self, calculation_method):
```

```
        """
        Updates calculation method
        :param calculation_method: methods: ['LEAST_SQUARES',
'LINEAR_INTERPOLATION']
        :return:
        """
        self.calculation_method = calculation_method

    def set_parameters(self, m, n):
        """
        Sets equation parameters
        :param m: grade 1 coefficient value in a linear equation
        :param n: constant coefficient value in a linear equation
        :return:
        """
        m, n = gt.to_number_n_dec(gt.N_DECIMALS, m, n)
        self.update_parameters(parameters_dictionary(m, n))

    def calculate_expression(self, point_1: list = None, point_2: list = None
):
        """
        Given at least 2 point, calculates the coefficients for a linear
expression
        :param point_1: first point
        :param point_2: second point
        :return:
        """
        match self.calculation_method:
            case 'LEAST_SQUARES':
                coefficients = np.polyfit(self.sort_x(), self.sort_y(), deg=1)
            case 'LINEAR_INTERPOLATION':
                coefficients = np.polyfit([point_1[0], point_2[0]], [point_1[1
], point_2[1]], deg=1)
            case _:
                raise ValueError('Calibration calculation method not accepted'
)
        self.set_parameters(coefficients[0], coefficients[1])

    def calculate_temperature(self, voltage: float):
        """
        Calculates the temperature with calibration equation rounded to 3
decimal points.
        :param voltage: Voltage value
        :return: Temperature value rounded to 3 decimal points
        """
        check_all_floats(voltage)
        return round(linear_func(voltage, self.get_parameter('coefficient_g1'
), self.get_parameter('constant')), 3)

    def plot_expression(self, axes, known_expression, x_plot=None):
        """
        Plots linear calibration graph on axes
        :param known_expression:
        :param x_plot:
        :param axes:
        :return:
        """
        x_list = x_plot if known_expression else self.sort_x()
        axes[0].plot(x_list, np.polyval([self.get_parameter('coefficient_g1'
), self.get_parameter('constant')],
                                        x_list), 'y-', label='Linear
```

```
Regression')

class NonLinearCalibration(Calibration):
    """
    Child class of calibration for the non-linear expression, here everything
related to linear calibration
    is managed and can convert to linear class if needed
    """
    def __init__(self):
        """
        Initiates object with its type
        """
        super().__init__("NON_LINEAR_EQUATION")

    def __repr__(self):
        """
        String representation method
        :return: calibration equation in string form
        """
        return f"y = {self.get_parameter('coefficient_g2'):.3f}x\u00B2" \
               f"{get_sign(self.get_parameter('coefficient_g1'))}" \
               f"{self.get_parameter('coefficient_g1'):.3f}x" \
               f"{get_sign(self.get_parameter('constant'))}" \
               f"{self.get_parameter('constant'):.3f}"

    def set_parameters(self, a: float, b: float, c: float):
        """
        Sets equation parameters
        :param a: grade 2 coefficient value in a non linear equation
        :param b: grade 1 coefficient value in a non linear equation
        :param c: constant coefficient value in a non linear equation
        :return:
        """
        check_all_floats(a, b, c)
        self.update_parameters(parameters_dictionary(a, b, c))

    def calculate_expression(self):
        """
        Calculated non-linear coefficient for the expression
        :return:
        """
        x_array = np.array(self.sort_x())
        y_array = np.array(self.sort_y())

        popt, _ = curve_fit(f=non_linear_func, xdata=x_array, ydata=y_array)

        a, b, c = popt
        self.set_parameters(a, b, c)

    def calculate_temperature(self, voltage: float):
        """
        Calculates the temperature with calibration equation rounded to 3
decimal points.
        :param voltage: Voltage value
        :return: Temperature value rounded to 3 decimal points
        """
        check_all_floats(voltage)
        return round(
            non_linear_func(voltage, self.get_parameter('coefficient_g2'),
self.get_parameter('coefficient_g1'),
                            self.get_parameter('constant')), 3)
```

```
    def plot_expression(self, axes, known_expression, x_plot=None):
        """
        Plots nonlinear calibration graph on axes
        :param known_expression: calibration expression
        :param x_plot: list of x values if there's an expression
        :param axes: plot information
        :return:
        """
        x_list = x_plot if known_expression else np.linspace(self.sort_x()[0
], self.sort_x()[-1], 100)
        y_plot = non_linear_func(x_list, self.get_parameter('coefficient_g2'
), self.get_parameter('coefficient_g1'),
                                 self.get_parameter('constant'))
        axes[0].plot(x_list, y_plot, 'y-', label='Fitted Curve')
```

src/daqTools.py

```python
import csv
import nidaqmx

import datetime as dt
import src.guiTools as gt

from nidaqmx.constants import (TerminalConfiguration)

# DAQ model list
modelsDAQ = ['USB-6211', 'USB-6001', 'USB-6002']

alarm_log_fieldnames = ['Alarm Type', 'Temperature', 'Time Interval']

AO_DAQ_NAME = "wheatstone_vcc"
AO_DAQ_MIN_VAL = 0
AO_DAQ_VAL = 1
AO_DAQ_MAX_VAL = 5


def is_daq_connected():
    system = nidaqmx.system.System.local()
    devices = system.devices
    return len(devices) > 0


class niDAQ:
    """
        Class with DAQ information.

        Attributes:
            model (string): DAQ model selected by the user.
        """

    def __init__(self, model, exit_requested):
        self.model = model
        self.task_ai_ao = []
        self.exit_requested = exit_requested
        self.calibration = ""
        self.calibrations_log = []
        self.alarm_min = None
        self.alarm_max = None
        self.sample_rate = None
        self.n_samples = None
        self.start_acquisition_time = ""
        self.data = []
        self.time_intervals = []
        self.alarms_log = []

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        """
        When object[index] is used, accesses data list to retrieve value
        :param index: index to access
        :return: pair with [voltage, temperature]
        """
        return self.data[index]
```

```
    def __repr__(self):
        return f"model: {self.model}, " \
               f"calibration: {repr(self.calibration)}, " \
               f"alarm: [min, max] = {[self.alarm_min, self.alarm_max]} ºC"


    def set_tasks(self):
        if is_daq_connected():
            for channel in range(2):
                self.task_ai_ao.append(nidaqmx.Task())
        else:
            raise ValueError("Number of devices found in system is 0")

    def set_exit_request(self):
        """
        Sets exit request to True
        :return: bool in True
        """
        self.exit_requested = True

    def set_alarm_min(self, alarm_min):
        """
        Sets minimum alarm parameter
        :param alarm_min:
        :return:
        """
        self.alarm_min = alarm_min

    def set_alarm_max(self, alarm_max):
        """
        Sets maximum alarm parameter
        :param alarm_max:
        :return:
        """
        self.alarm_max = alarm_max

    def set_calibration(self, expression):
        """
        Sets to true if user has assigned a calibration
        :return:
        """
        self.calibration = expression

    def set_time_log(self):
        """
        Sets time log to current date/month/year hour:minute:second
        :return:
        """
        self.start_acquisition_time = dt.datetime.now().strftime("%d/%m/%Y %H:
%M:%S.%f")

    def set_sample_rate(self, sample_rate):
        """
        Sets sample rate of data acquisition
        :param sample_rate: sample rate introduced by user
        :return:
        """
        self.sample_rate = sample_rate

    def set_n_samples(self, n_samples):
```

```
        """
        Sets number of samples in data acquisition
        :param n_samples: number of samples introduced by user
        :return:
        """
        self.n_samples = n_samples

    def get_sample_rate(self):
        """
        Sets sample rate of data acquisition
        :return: sample rate introduced by user
        """
        return self.sample_rate

    def get_n_samples(self):
        """
        Sets number of samples in data acquisition
        :return: number of samples introduced by user
        """
        return self.n_samples

    def get_time_log(self):
        """
        Returns moment when the data acquisition started
        :return: string "date/month/year hour:minute:second"
        """
        return self.start_acquisition_time

    def get_alarm_min(self):
        """
        Returns min alarm value
        :return: min temperature in [ºC]
        """
        return self.alarm_min

    def get_alarm_max(self):
        """
        Return max alarm value
        :return: max temperature in [ºC]
        """
        return self.alarm_max

    def get_calibration(self):
        """
        Returns the calibration object
        :return:
        """
        repr_calibration = [repr(calibrations) for calibrations in
self.calibrations_log]
        return self.calibrations_log[repr_calibration.index(self.calibration)]

    def disable_alarms(self):
        """
        Disables alarms
        :return:
        """
        self.alarm_max = None
        self.alarm_min = None

    def is_exit_requested(self):
        """
```

```
        Returns True if exit has been requested
        :return: bool
        """
        return self.exit_requested

    def has_data(self):
        """
        Returns true is there has been data collected
        :return: True: data > 0
        """
        return len(self) > 0

    def is_alarm_min_set(self):
        """
        Returns true if min alarm is set
        :return:
        """
        return self.alarm_min is not None

    def is_alarm_max_set(self):
        """
        Returns true if max alarm is set
        :return:
        """
        return self.alarm_max is not None

    def is_calibration_set(self):
        """
        Checks if there has been a calibration assigned
        :return: True if there has been
        """
        return self.calibration != ""

    def is_sampling_underway(self):
        """
        Checks if finite data acquisition is underway
        :return: True is the number of samples taken is smaller than the
number of samples introduced by the user
        """
        return len(self) < self.n_samples

    def calculate_time_interval_ms(self):
        """
        Calculates period in milliseconds given a sample_rate in Hertz
        :return: calculated period in milliseconds
        """
        if self.sample_rate == 0:
            raise ValueError("Sample rate cannot be zero.")
        return (1 / self.sample_rate) * 1000

    def read_voltage(self):
        """
        Simulates the reading of the voltage by the DAQ

        returns:
            voltage (float): reading of voltage by the DAQ
        """
        self.set_task_start(0)
        match self.model:
            case 'USB-6211':
                # simulation of temperature reading by the DAQ
```

```
                voltage = self.task_ai_ao[0].read()
            case 'USB-6001':
                # simulation of temperature reading by the DAQ

                voltage = self.task_ai_ao[0].read()
            case 'USB-6002':
                # simulation of temperature reading by the DAQ

                voltage = self.task_ai_ao[0].read()
            case _:
                raise ValueError(f"No matching model found.\nExpected:
{modelsDAQ}\nGot: {self.model}.")
        self.set_task_stop(0)
        return round(voltage, 3)

    def add_calibration_to_log(self, calibration):
        """
        Adds calibration parameters to log
        :param calibration: calibration object
        :return:
        """
        self.calibrations_log.append(calibration)

    def add_data(self, voltage_temperature: list, time_interval):
        """
        Given a voltage and temperature and time_interval, adds to data
        :param voltage_temperature: list [voltage, temperature]
        :param time_interval: time interval between sampling
        :return:
        """
        self.data.append(voltage_temperature)
        self.add_time(time_interval)

    def acquire_data(self, calibration, time_interval):
        """
        Reads voltage from DAQ, converts to temperature with calibration,
adds points to data
        :param time_interval:
        :param calibration:
        :return:
        """
        # reads voltage
        voltage = self.read_voltage()
        # calculates temperature
        temperature = calibration.calculate_temperature(voltage)
        # adds data
        self.add_data([voltage, temperature], time_interval)

    def add_time(self, time_interval):
        if not self.time_intervals:
            # If the list is empty, adds the first value starting from 0

            self.time_intervals.append(0)
        else:
            # If the list is not empty, adds the next value with the given
interval
            self.time_intervals.append(int(self.time_intervals[-1] +
time_interval))

    def add_alarms_log(self, is_min):
```

```
        alarm_entry = {
            'Alarm Type': 'Below Minimum' if is_min else 'Above Maximum',
            'Temperature': self[-1][1],
            'Time Interval': self.time_intervals[-1]
        }
        self.alarms_log.append(alarm_entry)

    def clear_data_acquisition(self):
        """
        Clears stored information from past logs like the data, parameters
and time
        :return:
        """
        self.data.clear()
        self.time_intervals.clear()
        self.alarms_log.clear()
        self.sample_rate = None
        self.n_samples = None
        self.start_acquisition_time = ""

    def save_data_acquisition(self, file_name):
        if not file_name.lower().endswith(".csv"):
            file_name += ".csv"
        with open(file_name, mode='w', newline='') as file:
            writer = csv.writer(file)

            # writes date and time
            writer.writerow([self.start_acquisition_time])
            writer.writerow([])

            # writes calibration
            writer.writerow(["CALIBRATION"])
            writer.writerow([self.calibration])
            writer.writerow([])

            # writes number of samples and sample rate

            writer.writerow(["PARAMETERS"])
            writer.writerow(["Number of samples", "Sample rate [Sa/s]"])
            writer.writerow([self.n_samples, self.sample_rate])
            writer.writerow([])

            # writes alarm logs
            writer.writerow(["ALARM LOGS"])
            writer.writerow(["Min alarm", "Max alarm"])
            writer.writerow([self.alarm_min, self.alarm_max])
            dic_writer = csv.DictWriter(file, fieldnames=alarm_log_fieldnames)
            dic_writer.writeheader()
            for entry in self.alarms_log:
                dic_writer.writerow(entry)
            writer.writerow([])

            # writes data
            writer.writerow(["DATA"])
            writer.writerow(["Voltage [V]", "Temperature [ºC]"])
            for voltage, temperature in self.data:
                writer.writerow([voltage, temperature])

    def generate_index_list(self):
        """
        Generates a list that goes from 1 to the number of data samples stored
```

```
        :return:
        """
        return [i for i in range(1, len(self.data) + 1)]

    def trigger_alarms(self, window, alarm_icon_keys):
        """
        Checks if alarms should be triggered and if so logs and triggers them
        :param window: gui window
        :param alarm_icon_keys: ['-MIN_TEMP_ICON-', '-MAX_TEMP_ICON-']
        :return:
        """
        if self.is_alarm_min_set():
            if self[-1][1] < self.get_alarm_min():
                self.add_alarms_log(is_min=True)
                window[alarm_icon_keys[0]].metadata = True
            else:
                window[alarm_icon_keys[0]].metadata = False
        if self.is_alarm_max_set():
            if self[-1][1] > self.get_alarm_max():
                self.add_alarms_log(is_min=False)
                window[alarm_icon_keys[1]].metadata = True
            else:
                window[alarm_icon_keys[1]].metadata = False

    def trigger_alarm_icon(self, window, alarm_icon_keys):
        # update min alarm image
        window[alarm_icon_keys[0]].update(
            source=gt.ALARM_MIN_ON_PATH if window[alarm_icon_keys[0
]].metadata else
            (gt.ALARM_MIN_OFF_PATH if self.is_alarm_min_set() else
gt.ALARM_UNSET_PATH))
        # update max alarm image
        window[alarm_icon_keys[1]].update(
            source=gt.ALARM_MAX_ON_PATH if window[alarm_icon_keys[1
]].metadata else
            (gt.ALARM_MAX_OFF_PATH if self.is_alarm_max_set() else
gt.ALARM_UNSET_PATH))

    def perform_data_acquisition(self, window, fig, figure_canvas_agg,
calibration, time_interval, alarm_icon_keys):
        self.acquire_data(calibration, time_interval)
        self.update_figure(fig, figure_canvas_agg)
        self.trigger_alarms(window, alarm_icon_keys)

    def update_figure(self, fig, figure_canvas_agg):
        axes = fig.axes  # getting the subplots
        axes[0].clear()
        axes[0].set_xlabel("Readings (ms)")
        axes[0].set_ylabel("Temperature (°C)")
        axes[0].grid()
        if self.has_data():
            self.plot_temperature_points(axes)
        if self.is_alarm_min_set() or self.is_alarm_max_set():
            self.plot_temperature_alarms(axes)

        figure_canvas_agg.draw()
        figure_canvas_agg.get_tk_widget().pack(side='top', fill='both',
expand=1)

    def plot_temperature_alarms(self, axes):
```

```
        x = [0, self.time_intervals[-1] if self.has_data() else 10]

        if self.is_alarm_min_set():
            y = [self.alarm_min] * 2
            axes[0].plot(x, y, 'b--')
        if self.is_alarm_max_set():
            y = [self.alarm_max] * 2
            axes[0].plot(x, y, 'r--')

    def plot_temperature_points(self, axes):
        x = self.time_intervals
        y = [temperature[1] for temperature in self.data]
        axes[0].plot(x, y, color='orange', linestyle='-')

    def set_task_start(self, index_ai_ao):
        self.task_ai_ao[index_ai_ao].start()

    def set_task_stop(self, index_ai_ao):
        self.task_ai_ao[index_ai_ao].stop()

    def set_task_write(self, value: float):
        self.task_ai_ao[1].write(value)

    def initiate_daq(self):
        self.set_tasks()
        # assignation of analog input
        self.add_analog_input()
        # assignation of analog output
        self.add_analog_output()

    def add_analog_input(self):
        """
        Defines analog input in DAQ
        :return:
        """
        self.task_ai_ao[0].ai_channels.add_ai_voltage_chan("Dev1/ai0",
terminal_config=TerminalConfiguration.DIFF)

    def add_analog_output(self):
        self.task_ai_ao[1].ao_channels.add_ao_voltage_chan("Dev1/ao0",
AO_DAQ_NAME,

min_val=AO_DAQ_MIN_VAL, max_val=AO_DAQ_MAX_VAL)

    def exit(self):
        print("Exit requested before calibration")
        self.set_task_stop(0)
        self.set_task_stop(1)
```

src/gui/guiCalibrationMethod.py

```
import src.guiTools as gt

from src.guiTools import sg

def get_layout_no_calibration():
    """
    Defines layout for the window when there is no calibration set
    :return: list with the layout
    """
    sg.theme(gt.DEFAULT_THEME)
    column = sg.Column([
        [sg.Push(), sg.Image(gt.ICON_PATH, size=(200, 200)), sg.Push()],
        [sg.Frame('Calibration Expression', [
            [sg.Text('No calibration yet...', pad=(10, 10))]
        ], element_justification='center', expand_x=True, pad=(10, 10),
relief=sg.RELIEF_RAISED)],
        [sg.Text('Choose Calibration Method:')],
        [sg.Radio(gt.CAL_METHOD_TEMP_VOLT, group_id="calMethod", k='-
TEMP_AND_VOLT-', default=True,
                  enable_events=True)],
        [sg.Radio(gt.CAL_METHOD_EXP, group_id="calMethod", k='-EXP-', default=
False, enable_events=True)],
        [sg.Push(),
         sg.Button('Calibrate', key="-CALIBRATE-"),
         sg.Button('Acquire Data', key='-DATA-', visible=False)]
    ], pad=((0, 0), (0, 110)))
    layout = [
        [sg.VPush()],
        [column],
        [sg.VPush()]
    ]
    return layout

def calibration_method_window(layout):
    """
    Creates window for the calibration menu
    :param layout: desired layout to show in the window
    :return: pysimplegui window
    """
    return sg.Window("Sensor Calibration", layout, size=(gt.WINDOW_WIDTH,
gt.WINDOW_HEIGHT),
                     element_justification='center')

def layout_with_expression(calibration_log):
    """
    Layout when there are already logged calibrations
    :param calibration_log: list of past calibrations
    :return: list with the layout
    """
    sg.theme(gt.DEFAULT_THEME)

    column = sg.Column([
        [sg.Push(), sg.Image(gt.ICON_PATH, size=(200, 200)), sg.Push()],
        [sg.Frame('Calibration Expression', [
            [sg.Combo(calibration_log,
                      default_value=calibration_log[0],
                      key='-CALIBRATIONS_LOG-',
                      expand_x=True,
```

```
                      enable_events=True,
                      pad=(10, 10))],
        ], element_justification='center', expand_x=True, pad=(10, 10),
relief=sg.RELIEF_RAISED)],
        [sg.Text('Choose Calibration Method:')],
        [sg.Radio(gt.CAL_METHOD_TEMP_VOLT, group_id="calMethod", k='-
TEMP_AND_VOLT-', default=True,
                  enable_events=True)],
        [sg.Radio(gt.CAL_METHOD_EXP, group_id="calMethod", k='-EXP-', default=
False, enable_events=True)],
        [sg.Push(),
         sg.Button('Calibrate', key="-CALIBRATE-"),
         sg.Button('Acquire Data', key='-DATA-', visible=True)]
    ], pad=((0, 0), (0, 110)))

    layout = [
        [sg.VPush()],
        [column],
        [sg.VPush()]
    ]
    return layout

def run_calibration_method_no_calibration_window(window):
    """
    Runs pysimplegui window behavior when there is no calibration set
    :param window: pysimplegui window
    :return: string with the option selected by user['EXIT',
'EXPRESSION_INPUT', 'ACQUIRE_DATA']
    """

    while True:
        event, values = window.read()

        if event == sg.WIN_CLOSED:
            window.close()
            return 'EXIT'

        if event == '-CALIBRATE-':
            if values['-TEMP_AND_VOLT-']:
                window.close()
                return 'TEMP_VOLTAGE'
            elif values['-EXP-']:
                window.close()
                return 'EXPRESSION_INPUT'

        if event == '-DATA-':
            window.close()
            return 'ACQUIRE_DATA'


def run_calibration_method_window(window, niDAQ):
    """
    Runs pysimplegui window behavior when there are calibrations logged
    :param window: pysimplegui window
    :param niDAQ: object with past calibrations
    :return: string with the option selected by user['EXIT',
'EXPRESSION_INPUT', 'ACQUIRE_DATA']
    """
    while True:
        event, values = window.read()  # window.read returns event and values
```

```
if event == sg.WIN_CLOSED:
    window.close()
    return 'EXIT'

if event == '-CALIBRATIONS_LOG-':
    niDAQ.set_calibration(values['-CALIBRATIONS_LOG-'])

if event == '-CALIBRATE-':
    if values['-TEMP_AND_VOLT-']:
        window.close()
        return 'TEMP_VOLTAGE'
    elif values['-EXP-']:
        window.close()
        return 'EXPRESSION_INPUT'

if event == '-DATA-':
    window.close()
    return 'ACQUIRE_DATA'
```

src/gui/guiDAQ.py

```
import src.guiTools as gt

from src.guiTools import sg

def select_daq_window(modelDAQ):
    """
    Creates a window so that the user chooses their DAQ model

    Arguments:
        modelDAQ (str list): DAQ model list

    Returns:
        model: (str): DAQ model that the user has chosen
    """
    sg.theme(gt.DEFAULT_THEME)
    column = sg.Column([
        [sg.Push(), sg.Image(gt.ICON_PATH, size=(200, 200)), sg.Push()],
        [sg.Text('Select the model of the National Instruments DAQ:', pad=((0
, 0), (15, 0)))],
        [sg.Combo(modelDAQ,
                  default_value="Select the model...",
                  key='-MODEL-',
                  expand_x=True,
                  tooltip='Select an option before moving forward')],
        [sg.Push(), sg.Button('OK', key='-OK-', bind_return_key=True)]
    ], pad=((0, 0), (0, 120)))
    layout = [
        [sg.VPush()],
        [column],
        [sg.VPush()]
    ]

    window = sg.Window('PyroDAQ', layout, size=(gt.WINDOW_WIDTH,
gt.WINDOW_HEIGHT), element_justification='center')

    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED:
            window.close()
            return None, True

        elif event == '-OK-':
            window['-MODEL-'].set_tooltip("")
            model = values['-MODEL-']
            window.close()
            return model, False


def no_daq_detected_popup(e):
    """
    Popup error window that warns there is no DAQ detected
    :param e: ValueError from trying to initiate DAQ
    :return:
    """
    sg.theme(gt.ACCENT_THEME)
    sg.Window('Error', [[sg.Text(f'No DAQ detected:\n{e}')]]).read()
```

src/gui/guiDataAcquisition.py

```
import src.guiTools as gt
from src.guiTools import sg

alarm_input_keys = ['-MIN_TEMP_INPUT-', '-MAX_TEMP_INPUT-']
alarm_icon_keys = ['-MIN_ALARM_ICON-', '-MAX_ALARM_ICON-']
parameters_input_keys = ['-N_SAMPLES_INPUT-', '-SAMPLE_RATE_INPUT-']

def data_acquisition_window(calibration_expression):
    """
    Pysimplegui layout and window for data acquisition
    :param calibration_expression: string with current calibration expression
    :return: pysimplegui window with relevant layout
    """
    sg.theme(gt.DEFAULT_THEME)

    first_column = sg.Column([
        [sg.Button('Recalibrate', k='-RECALIBRATE-')],
        [sg.VPush()],
        [sg.Frame('Calibration', [
            [sg.Text(calibration_expression, pad=(10, 10))]
        ], element_justification='center', expand_x=True, pad=(10, 10),
relief=sg.RELIEF_RAISED)],
        [sg.Frame('Temperature Alarms', [
            [sg.Column([
                [sg.Text('Min ='),
                 sg.Input(size=gt.SIZE_INPUT, key='-MIN_TEMP_INPUT-',
enable_events=True),
                 sg.Text(' [°C]')],
                [sg.Image(gt.ALARM_UNSET_PATH, key='-MIN_ALARM_ICON-',
metadata=False)],
                [sg.Text('Unset', key='-MIN_TEMP_TXT-')]
            ], element_justification='center'),
                sg.Column([
                    [sg.Text('Max ='),
                     sg.Input(size=gt.SIZE_INPUT, key='-MAX_TEMP_INPUT-',
enable_events=True),
                     sg.Text(' [°C]')],
                    [sg.Image(gt.ALARM_UNSET_PATH, key='-MAX_ALARM_ICON-',
metadata=False)],
                    [sg.Text('Unset', key='-MAX_TEMP_TXT-')]
                ], element_justification='center'),
                sg.Column([
                    [sg.Push()]
                ]),
                sg.Column([
                    [sg.VPush()],
                    [sg.Push(), sg.Button('Set', k='-SET-')],
                    [sg.Push(), sg.Button('Disable', k='-DISABLE-', visible=
False)]
                ], element_justification='right')
            ]
        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
        [sg.Frame('Choose Data Acquisition Type:', [
            [sg.Radio(gt.DATA_ON_DEMAND,
                      group_id='acq_type',
                      default=True,
                      k='-ON_DEMAND-',
                      enable_events=True,
```

```
                        pad=((10, 0), (10, 0)))],
              [sg.Radio(gt.DATA_CUSTOM, group_id='acq_type', k='-
FINITE_SAMPLING-', enable_events=True,
                        pad=((10, 0), (10, 0)))],
              [sg.Text('No. Samples:', pad=((40, 0), 0)),
               sg.Input(size=gt.SIZE_INPUT, key='-N_SAMPLES_INPUT-', disabled=
True, enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1])],
              [sg.Text('Sample Rate:', pad=((40, 0), (0, 10))),
               sg.Input(size=gt.SIZE_INPUT, key='-SAMPLE_RATE_INPUT-', disabled=
True, enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1], pad=(0, (0, 10
))),
               sg.Text('Sa/s', pad=((0, 10), (0, 10)))]
          ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
          [sg.Push(), sg.Button('Acquire Data', k='-ACQUIRE-', metadata=False)],
          [sg.Frame('Time Interval [ms]', [
              [sg.Slider(range=(gt.MIN_TIME_UPDATE_MS,
gt.MAX_TIME_INTERVAL_MS), default_value=500, resolution=10,
                         orientation='h', key='-SLIDER-', size=(40, 15),
tick_interval=1000)]
          ], key='-TIME_INTERVAL-', visible=False, expand_x=True, pad=(10, 10),
element_justification='center',
                    relief=sg.RELIEF_SUNKEN)],
          [sg.VPush()]

      ], expand_x=True, expand_y=True)

      second_column = sg.Column([
          [sg.Push(),
           sg.Text("Samples Collected: ", key='-SAMPLES_COLLECTED_TXT-',
visible=False),
           sg.Text("", key='-SAMPLES_COLLECTED_VALUE-', size=gt.SIZE_INPUT,
visible=False)],
          [sg.Canvas(k='-CANVAS-', size=(200, 200))],
          [sg.Button('Stop', k='-STOP-', visible=False, pad=(10, 10)),
           sg.Button('Reset', k='-RESET-', visible=False, pad=(10, 10))],
          [sg.Push(), sg.Button('Save Data', k='-SAVE-', visible=False)]
      ], expand_x=True, element_justification='center')

      layout = [
          [sg.VPush()],
          [first_column, second_column],
          [sg.VPush()]
      ]
      return gt.gui_window_with_graph('Data Acquisition', layout,
gt.FIG_SIZE_WIDTH, gt.FIG_SIZE_HEIGHT, False)

  def data_acquisition_window_behavior(niDAQ, window, fig, figure_canvas_agg):
      """
      Data acquisition window behavior
      :param niDAQ: object where data will be stored
      :param window: pysimplegui window with data acquisition layout
      :param fig: data plot
      :param figure_canvas_agg: canvas for the data plot
      :return:
      """
      time_interval = None
      min_frequency = gt.calculate_frequency(gt.MAX_TIME_INTERVAL_MS) * 1000
```

```
    max_frequency = gt.calculate_frequency(gt.MIN_TIME_UPDATE_MS) * 1000

    while True:
        event, values = window.read(timeout=time_interval)
        if event == sg.WIN_CLOSED:
            niDAQ.set_exit_request()
            break

        if event == '-RECALIBRATE-':
            break

        # only accepts digits, decimal point '.' and '-'

        if event in alarm_input_keys:
            gt.filter_numeric_characters(window, values, event,
alarm_input_keys)

        # only accepts digits
        if event in parameters_input_keys:
            gt.filter_digits(window, values, event, ['-N_SAMPLES_INPUT-'])
            gt.filter_numeric_characters(window, values, event, ['-
SAMPLE_RATE_INPUT-'])

        if event == '-SET-':
            try:
                # checks if both inputs are empty

                if all(values[key] == "" for key in alarm_input_keys):
                    raise ValueError("Values must be assigned")
                elif all(values[key] != "" for key in alarm_input_keys):
                    alarm_min, alarm_max = gt.to_number_n_dec(gt.N_DECIMALS,
values['-MIN_TEMP_INPUT-'],
                                                              values['-
MAX_TEMP_INPUT-'])
                    if alarm_min >= alarm_max:
                        raise ValueError("Min alarm can't be bigger or equal
to max alarm")
                    niDAQ.set_alarm_min(alarm_min)
                    niDAQ.set_alarm_max(alarm_max)
                else:
                    if values['-MIN_TEMP_INPUT-'] != "":
                        [alarm_min] = gt.to_number_n_dec(gt.N_DECIMALS,
values['-MIN_TEMP_INPUT-'])
                        if niDAQ.is_alarm_max_set() and (alarm_min >=
niDAQ.get_alarm_max()):
                            raise ValueError("Min alarm can't be bigger or
equal to already set max alarm")
                        else:
                            niDAQ.set_alarm_min(alarm_min)
                    if values['-MAX_TEMP_INPUT-'] != "":
                        [alarm_max] = gt.to_number_n_dec(gt.N_DECIMALS,
values['-MAX_TEMP_INPUT-'])
                        if niDAQ.is_alarm_min_set() and (alarm_max <=
niDAQ.get_alarm_min()):
                            raise ValueError("Max alarm can't be bigger or
equal to already set min alarm")
                        else:
                            niDAQ.set_alarm_max(alarm_max)
                niDAQ.update_figure(fig, figure_canvas_agg)
                niDAQ.trigger_alarm_icon(window, alarm_icon_keys)
                gt.set_visible(window, True, '-DISABLE-')
```

```
            except Exception as e:
                sg.popup_error(str(e), title="Error")

            gt.empty_inputs(window, '-MIN_TEMP_INPUT-', '-MAX_TEMP_INPUT-')

        if event == '-DISABLE-':
            niDAQ.disable_alarms()
            window[alarm_icon_keys[0]].metadata = False
            window[alarm_icon_keys[1]].metadata = False
            niDAQ.trigger_alarm_icon(window, alarm_icon_keys)
            niDAQ.update_figure(fig, figure_canvas_agg)
            gt.set_visible(window, False, '-DISABLE-')

        if event == '-ON_DEMAND-':
            gt.set_disabled(window, True, '-N_SAMPLES_INPUT-', '-
SAMPLE_RATE_INPUT-')

        if event == '-FINITE_SAMPLING-':
            gt.set_disabled(window, False, '-N_SAMPLES_INPUT-', '-
SAMPLE_RATE_INPUT-')

        if event == '-ACQUIRE-':
            # saves moment in time when acquisition starts

            niDAQ.clear_data_acquisition()
            niDAQ.set_time_log()
            # if on demand data acquisition is selected

            if values['-ON_DEMAND-']:
                # from not reading to on demand
                window['-ACQUIRE-'].metadata = True
                gt.set_visible(window, True, '-STOP-', '-TIME_INTERVAL-')
                gt.set_visible(window, False, '-SAVE-')
                gt.set_disabled(window, True, '-FINITE_SAMPLING-')
            elif values['-FINITE_SAMPLING-']:
                try:
                    [sample_rate] = gt.check_if_valid_input(values,
gt.N_DECIMALS, '-SAMPLE_RATE_INPUT-')
                    [n_samples] = gt.check_if_valid_input(values, 0, '-
N_SAMPLES_INPUT-')
                    if not min_frequency <= sample_rate <= max_frequency:
                        raise ValueError(f"Sample rate must be between"

                                         f" {min_frequency:.3f} and
{max_frequency:.3f} Sa/s."
                                         f"\nGot {sample_rate} instead.")
                    elif not 2 <= n_samples <= 10000:
                        raise ValueError(f"Number of samples must be between
2 and 10k.\n"
                                         f"Got {n_samples} instead.")
                    else:
                        niDAQ.set_sample_rate(sample_rate)
                        niDAQ.set_n_samples(n_samples)
                        # from not reading to finite sampling

                        window['-ACQUIRE-'].metadata = True
                        # sets time_interval
                        time_interval = niDAQ.calculate_time_interval_ms()
                        gt.set_visible(window, True, '-STOP-')
                        gt.set_visible(window, False, '-SAVE-', '-ACQUIRE-')
                except Exception as e:
```

```
                    sg.popup_error(str(e), title="Error")
                    gt.empty_inputs(window, '-SAMPLE_RATE_INPUT-', '-
N_SAMPLES_INPUT-')

        if event == '-STOP-':
            window['-ACQUIRE-'].metadata = False
            gt.set_visible(window, False, '-STOP-')
            gt.set_visible(window, True, '-RESET-', '-ACQUIRE-')
            if values['-ON_DEMAND-']:
                gt.set_disabled(window, False, '-FINITE_SAMPLING-')
            if values['-FINITE_SAMPLING-']:
                gt.set_visible(window, True, '-SAVE-')

        if event == '-SAVE-':
            try:
                file_name = sg.popup_get_file("Save CSV File",
default_path=gt.get_desktop_dir(),
                                              default_extension="*.csv",
save_as=True,
                                              file_types=(("CSV Files",
"*.csv"),))
                if file_name == '' or file_name[-1] == '/':
                    raise ValueError("File name can't be empty.")
                elif file_name is not None:
                    niDAQ.save_data_acquisition(file_name)
                else:
                    raise ValueError("Couldn't save file.")
                sg.popup("Success", "Data saved to CSV successfully!")
            except Exception as e:
                sg.popup_error(str(e), title="Error")

        if event == '-RESET-':
            niDAQ.clear_data_acquisition()
            niDAQ.update_figure(fig, figure_canvas_agg)
            gt.set_visible(window, False, '-RESET-', '-SAVE-', '-
SAMPLES_COLLECTED_TXT-', '-SAMPLES_COLLECTED_VALUE-')

        window['-MIN_TEMP_TXT-'].update(f"{niDAQ.get_alarm_min()} [ºC]" if
niDAQ.is_alarm_min_set() else 'Unset')
        window['-MAX_TEMP_TXT-'].update(f"{niDAQ.get_alarm_max()} [ºC]" if
niDAQ.is_alarm_max_set() else 'Unset')

        if window['-ACQUIRE-'].metadata:
            gt.set_disabled(window, True, '-N_SAMPLES_INPUT-', '-
SAMPLE_RATE_INPUT-')
            gt.set_visible(window, False, '-RESET-', '-ACQUIRE-')
            gt.set_visible(window, True, '-SAMPLES_COLLECTED_TXT-', '-
SAMPLES_COLLECTED_VALUE-')
            if values['-ON_DEMAND-']:
                time_interval = values['-SLIDER-']
                niDAQ.perform_data_acquisition(window, fig, figure_canvas_agg,
                                               niDAQ.get_calibration(),
time_interval, alarm_icon_keys)
            elif values['-FINITE_SAMPLING-']:
                if niDAQ.is_sampling_underway():
                    niDAQ.perform_data_acquisition(window, fig,
figure_canvas_agg,
                                                   niDAQ.get_calibration(),
time_interval, alarm_icon_keys)
                else:
                    window['-ACQUIRE-'].metadata = False
```

```
                    gt.set_visible(window, True, '-RESET-', '-SAVE-', '-
ACQUIRE-')
                    gt.set_visible(window, False, '-STOP-')
            else:
                raise ValueError("Acquiring data incorrectly")
            window['-SAMPLES_COLLECTED_VALUE-'].update(len(niDAQ))
            niDAQ.trigger_alarm_icon(window, alarm_icon_keys)

        else:
            gt.set_disabled(window, False, '-ACQUIRE-')
            time_interval = None
            if values['-ON_DEMAND-']:
                gt.set_visible(window, False, '-TIME_INTERVAL-')
            if values['-FINITE_SAMPLING-']:
                gt.set_disabled(window, False, '-N_SAMPLES_INPUT-', '-
SAMPLE_RATE_INPUT-')

    window.close()
```

src/gui/guiExpressionInputCalibrate.py

```
import src.calibrationTools as ct
import src.guiTools as gt
from src.guiTools import sg
import time


text_input_keys = ['-A_INPUT-', '-B_INPUT-', '-C_INPUT-', '-M_INPUT-', '-
N_INPUT-']

def expression_calibrate_window():
    """
    Window for direct expression
    :return: pysimplegui window with its layout
    """
    sg.theme(gt.DEFAULT_THEME)
    first_column = sg.Column([
        [sg.Frame('Choose Expression Type:', [
            [sg.Radio(gt.TEMP_VOLT_LIN_EQ,
                      group_id='exp_type',
                      default=True,
                      k='-LINEAR_EQ-',
                      enable_events=True,
                      pad=((10, 0), (10, 0)))],
            [sg.Text('m =', pad=((40, 0), 0), ),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-M_INPUT-',
                      enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
             sg.Text('n ='),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-N_INPUT-',
                      enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1])],
            [sg.Radio(gt.TEMP_VOLT_NON_LINEAR_EQ,
                      group_id='exp_type',
                      default=False,
                      k='-NON_LINEAR_EQ-',
                      enable_events=True,
                      pad=((10, 0), (10, 10)))],
            [sg.Text('a =', pad=((40, 0), 0), ),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-A_INPUT-',
                      enable_events=True,
                      disabled=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
             sg.Text('b ='),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-B_INPUT-',
                      enable_events=True,
                      disabled=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
             sg.Text('c ='),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-C_INPUT-',
```

```
                    enable_events=True,
                    disabled=True,

disabled_readonly_background_color=sg.theme_button_color()[1])]
        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
        [sg.Push(), sg.Button('Enter', k='-ENTER-')],
        [sg.Frame('Calculated expression:', [
            [sg.Text("Calculating expression...", pad=(10, 10), k='-EQUATION-'
),
             sg.Button("Copy", key="-COPY-", tooltip="Copy to clipboard",
pad=((10, 10), (10, 10)), visible=False)],
        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_RAISED,
element_justification='center')],
        [sg.Frame('Voltage Input:', [
            [sg.Text("Type In", pad=(10, 0), k='-TOGGLE_OFF_TXT-',
font=gt.FONT_BOLD),
             sg.Button(image_filename=gt.TOGGLE_OFF_PATH,
                       key='-TOGGLE-',
                       button_color=(sg.theme_background_color(),
sg.theme_background_color()),
                       border_width=0,
                       metadata=False),
             sg.Text("Measure", k='-TOGGLE_ON_TXT-', font=gt.FONT_DEFAULT)],
            [sg.Text('V =', k='-V_TXT-', pad=(10, 10)),
             sg.Input(size=gt.SIZE_INPUT,
                      key='-V_INPUT-',
                      enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
             sg.Text('[V]')]
        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
        [sg.Push(), sg.Button('Calculate', k='-CALCULATE-', disabled=True)]
    ])

    second_column = sg.Column(
        [
            [sg.Canvas(k='-CANVAS-', size=(200, 200))],
            [sg.Table(values=[],
                      headings=['Voltage (V)', 'Temperature (°C)'],
                      k='-TABLE-',
                      num_rows=5,
                      enable_click_events=True,
                      enable_events=True,
                      expand_x=True)],
            [sg.Button('Clear', k='-CLEAR-', tooltip=" Clear table ",
disabled=True),
             sg.Button('Delete', k='-DELETE-', tooltip=" Delete last row ",
disabled=True)],
            [sg.Push(), sg.Button('Choose', k='-CHOOSE-', disabled=True)]
        ]
    )

    # Define the layout
    layout = [
        [sg.VPush()],
        [first_column, second_column],
        [sg.VPush()]
    ]

    return gt.gui_window_with_graph('Input Sensor Calibration Equation',
layout,
```

```
                                    gt.FIG_SIZE_WIDTH, gt.FIG_SIZE_HEIGHT,
False)

def expression_input_calibrate_window_behavior(niDAQ, window, fig,
figure_canvas_agg):
    """
    Behaviour for direct expression input calibration window
    :param niDAQ: object
    :param window: pysimplegui window
    :param fig: calibration plot
    :param figure_canvas_agg: canvas for calibration plot
    :return: calibration object
    """
    calibration = ct.LinearCalibration()
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED:
            break

        if event == '-LINEAR_EQ-':
            gt.set_disabled(window, False, '-M_INPUT-', '-N_INPUT-')
            window['-A_INPUT-'].update('', disabled=True)
            window['-B_INPUT-'].update('', disabled=True)
            window['-C_INPUT-'].update('', disabled=True)

        if event == '-NON_LINEAR_EQ-':
            window['-M_INPUT-'].update('', disabled=True)
            window['-N_INPUT-'].update('', disabled=True)
            gt.set_disabled(window, False, '-A_INPUT-', '-B_INPUT-', '-
C_INPUT-')

        if event == '-CHOOSE-':
            window.close()
            return calibration

        # only accepts digits, decimal point '.' and '-'

        if event in text_input_keys:
            gt.filter_numeric_characters(window, values, event,
text_input_keys)

        if event == '-ENTER-':
            try:
                if values['-LINEAR_EQ-']:
                    # checks if any input value is empty

                    if any(values[key] == '' for key in ['-M_INPUT-', '-
N_INPUT-']):
                        raise ValueError("Values must be assigned")
                    # checks if the input is a valid number

                    if any(not gt.is_number(values[key]) for key in ['-
M_INPUT-', '-N_INPUT-']):
                        raise ValueError("Values must be a numeric value.")
                    # checks if equation type has changed

                    if calibration.is_type('LINEAR_EQUATION'):
                        # updates parameters
                        calibration.set_parameters(values['-M_INPUT-'],
values['-N_INPUT-'])
                    else:
```

```
                        # changes object from nonlinear to linear calibration

                        calibration =
calibration.to_linear_calibration(values['-M_INPUT-'], values['-N_INPUT-'])

                elif values['-NON_LINEAR_EQ-']:
                    # checks if any input is empty

                    if any(values[key] == '' for key in ['-A_INPUT-', '-
B_INPUT-', '-C_INPUT-']):
                        raise ValueError("Values must be assigned")
                    # checks if the input is a valid number

                    if not any(gt.is_number(values[key]) for key in ['-
A_INPUT-', '-B_INPUT-', '-C_INPUT-']):
                        raise ValueError("Values must be a numeric value.")
                    # checks if equation type has changed

                    if calibration.is_type('NON_LINEAR_EQUATION'):
                        # updates parameters
                        a, b, c = gt.to_number_n_dec(gt.N_DECIMALS,
                                                     values['-A_INPUT-'],
values['-B_INPUT-'], values['-C_INPUT-'])
                        calibration.set_parameters(a, b, c)
                    else:
                        # changes object from nonlinear to linear calibration

                        calibration =
calibration.to_nonlinear_calibration(values['-A_INPUT-'], values['-B_INPUT-'],

values['-C_INPUT-'])

                window['-EQUATION-'].update(value=repr(calibration))
                gt.set_disabled(window, False, '-CHOOSE-', '-CALCULATE-')
                gt.set_visible(window, True, '-COPY-')
                # empties text inputs
                for key in text_input_keys:
                    window[key].update('')
                calibration.update_data()
                window['-TABLE-'].update(values=calibration.data)
                calibration.update_figure(fig, figure_canvas_agg,
known_expression=True)
            except ValueError as e:
                sg.popup_error(str(e), title="Error")

        if isinstance(event, tuple):
            # TABLE CLICKED Event has value in format ('-TABLE=',
'+CLICKED+', (row,col))
            # You can also call Table.get_last_clicked_position to get the
cell clicked
            if event[0] == '-TABLE-':
                if event[2][0] not in [-1, None]:  # If an actual row was
clicked
                    calibration.update_figure(fig, figure_canvas_agg,
                                              known_expression=True,
                                              is_point_selected=True,
                                              x_sel_point=calibration[event[2]
[0]][0],
                                              y_sel_point=calibration[event[2]
[0]][1])
```

```
        if event == '-DELETE-':
            del calibration[-1]
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=True)
            calibration.update_figure(fig, figure_canvas_agg,
known_expression=True)

        if event == '-CLEAR-':
            calibration.clear_data()
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=True)
            calibration.update_figure(fig, figure_canvas_agg,
known_expression=True)

        if event == '-COPY-':
            window['-COPY-'].update('Text Copied!', disabled=True)
            sg.clipboard_set(repr(calibration))  # Copy the text to clipboard

            time.sleep(1)
            window['-COPY-'].update('Copy', disabled=False)

        if event == '-TOGGLE-':
            gt.gui_toggle_behaviour(window)

        if event == '-CALCULATE-':
            try:
                if not window['-TOGGLE-'].metadata:
                    if values['-V_INPUT-'] == "":
                        raise ValueError("Values must be assigned")
                    elif not gt.is_number(values['-V_INPUT-']):
                        raise ValueError("Values must be a numeric value.")
                    inputVoltage = float(values['-V_INPUT-'])
                    if inputVoltage in calibration.data:
                        raise ValueError("Data input is repeated.")
                else:
                    inputVoltage = niDAQ.read_voltage()

                calibration.add_voltage(inputVoltage)

            except ValueError as e:
                sg.popup_error(str(e), title="Error")

            calibration.update_figure(fig, figure_canvas_agg,
known_expression=True)
            window['-TABLE-'].update(values=calibration.data)
            window['-V_INPUT-'].update('')

        if len(calibration) > 0:
            gt.set_disabled(window, False, '-CLEAR-', '-DELETE-')
        else:
            gt.set_disabled(window, True, '-CLEAR-', '-DELETE-')

    window.close()
```

src/gui/guiTempVoltCalibrate.py

```
import src.calibrationTools as ct
import time

import src.guiTools as gt

from src.guiTools import sg

text_input_keys = ['-V_INPUT-', '-T_INPUT-']

def select_points_window(data: list):
    """
    Behaviour for window with layout for selecting points in the
interpolation method
    :param data: pair list with data points
    :return: list with two data points selected
    """
    sg.theme(gt.ACCENT_THEME)

    data_sorted = gt.sort_pair_list_by_x(list(data))
    available_points = list(data_sorted)

    column_left = sg.Column([
        [sg.Canvas(key='-CANVAS-')]
    ])

    column_right = sg.Column([
        [sg.Table(values=data_sorted,
                  headings=['Voltage (V)', 'Temperature (ºC)'],
                  k='-TABLE-',
                  selected_row_colors='green on white',
                  enable_click_events=True)],
        [sg.Push(), sg.Button('Select', k='-SELECT-')],
        [sg.Frame('Selected Points: ', [
            [sg.Text('list of points...', k='-POINTS-')]
        ], expand_x=True)],
        [sg.Push(), sg.Button("Choose", k='-CHOOSE-', visible=False)]
    ])

    layout = [
        [sg.VPush()],
        [column_left, column_right],
        [sg.VPush()]
    ]

    window, fig, figure_canvas_agg = gt.gui_window_with_graph("Choose Points"
, layout,

gt.FIG_SIZE_WIDTH, gt.FIG_SIZE_HEIGHT, True)

    axes, x, y = gt.get_axes_for_points(fig, data_sorted)
    gt.draw_points(axes, x, y, 'bo', "Data Points")
    axes[0].legend()
    gt.pack_canvas(figure_canvas_agg)

    selected_points = []

    while True:
        event, values = window.read()
```

```
        axes, x, y = gt.get_axes_for_points(fig, data_sorted)
        gt.draw_points(axes, x, y, 'bo', "Data Points")

        if event == sg.WIN_CLOSED:
            break

        if event == '-CHOOSE-':
            window.close()
            return gt.sort_pair_list_by_x(selected_points)

        if isinstance(event, tuple):
            # TABLE CLICKED Event has value in format ('-TABLE=',
'+CLICKED+', (row,col))
            if event[0] == '-TABLE-':
                if event[2][0] not in [-1, None]:  # Header was clicked and
wasn't the "row" column
                    gt.draw_points(axes, available_points[event[2][0]][0],
available_points[event[2][0]][1], 'yo')

        if event == '-SELECT-':
            if values['-TABLE-']:
                selected_row = values['-TABLE-'][0]

                if len(selected_points) >= 2:
                    available_points.append(selected_points[0])
                    selected_points.pop(0)  # Remove the oldest point

                selected_points.append(available_points[selected_row])  # Add
the selected point
                available_points.pop(selected_row)
                available_points = gt.sort_pair_list_by_x(available_points)

                window['-TABLE-'].update(values=available_points)  # Clear
the table selection
                window['-POINTS-'].update(selected_points)

        if selected_points:
            gt.draw_points(axes, gt.get_sorted_nth_elements(selected_points, 0
),
                           gt.get_sorted_nth_elements(selected_points, 1), 'r-
o', "Selected Points")
            if len(selected_points) > 1:
                gt.set_visible(window, True, '-CHOOSE-')

        axes[0].legend()
        gt.pack_canvas(figure_canvas_agg)

    window.close()

def temp_volt_calibrate_window():
    """
    Window and layout for temperature-voltage relation
    :return: window with layout
    """
    sg.theme(gt.DEFAULT_THEME)

    column_left = sg.Column([
                [sg.Frame('Choose Expression Type:', [
                    [sg.Radio(gt.TEMP_VOLT_LIN_EQ,
                              group_id='exp_type',
```

```
                                default=True,
                                k='-LINEAR_EQ-',
                                enable_events=True,
                                pad=((10, 0), (10, 0)))],
                            [sg.Radio(gt.TEMP_VOLT_LEAST_SQUARES,
                                pad=((40, 0), 0),
                                group_id='lin_eq',
                                default=True,
                                enable_events=True,
                                k='-LEAST_SQUARES-')],
                            [sg.Radio(gt.TEMP_VOLT_LIN_INTERP,
                                pad=((40, 0), 0),
                                group_id='lin_eq',
                                default=False,
                                enable_events=True,
                                k='-LINEAR_INTERPOLATION-'),
                             sg.Button("Choose Points",
                                 k='-CHOOSE_POINTS-',
                                 visible=False,
                                 pad=((10, 0), 0), )],
                            [sg.Radio(gt.TEMP_VOLT_NON_LINEAR_EQ,
                                group_id='exp_type',
                                default=False,
                                k='-NON_LINEAR_EQ-',
                                enable_events=True,
                                pad=((10, 0), (10, 10)))]
                        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
                        [sg.Frame('Input Data:', [
                            [sg.Text("Type In", pad=(10, 0), k="-TOGGLE_OFF_TXT-",
font=gt.FONT_BOLD),
                             sg.Button(image_filename=gt.TOGGLE_OFF_PATH,
                                 key='-TOGGLE-',
                                 button_color=(sg.theme_background_color(),
sg.theme_background_color()),
                                 border_width=0,
                                 metadata=False),
                             sg.Text("Measure", k="-TOGGLE_ON_TXT-")],
                            [sg.Text('V =', k='-V_TXT-', pad=(10, 0)),
                             sg.Input(size=gt.SIZE_INPUT,
                                key='-V_INPUT-',
                                enable_events=True,

disabled_readonly_background_color=sg.theme_button_color()[1]),
                             sg.Text('T ='),
                             sg.Input(size=gt.SIZE_INPUT, key='-T_INPUT-',
enable_events=True),
                             sg.Button('Enter', k='-ENTER-', bind_return_key=True,
pad=((10, 0), (10, 10)))]
                        ], expand_x=True, pad=(10, 10), relief=sg.RELIEF_SUNKEN)],
                        [sg.Table(values=[],
                                  headings=['Voltage (V)', 'Temperature (ºC)'],
                                  k='-TABLE-',
                                  enable_click_events=True,
                                  enable_events=True)],
                        [sg.Text('Number of Samples: '),
                         sg.Text('0', k='-N_SAMPLES-'),
                         sg.Push(),
                         sg.Button('Clear', k='-CLEAR-', tooltip=" Clear table ",
disabled=True),
                         sg.Button('Delete', k='-DELETE-', tooltip=" Delete last row "
, disabled=True)]
```

```
            ])

    column_right = sg.Column([
                [sg.Frame('Calibration Equation',
                          [[sg.Text('Expression...',
                                    k='-EQ_EXPRESSION-',
                                    enable_events=True,
                                    metadata=False,
                                    pad=(10, 10)),
                            sg.Button("Copy", key="-COPY-", tooltip="Copy to
clipboard", pad=(10, 10), visible=False)]
                           ],
                          expand_x=True,
                          expand_y=True,
                          pad=(10, 10),
                          element_justification='center',
                          relief=sg.RELIEF_RAISED)],
                [sg.Canvas(k='-CANVAS-', size=(200, 200))],
                [sg.Push(), sg.Button('Choose', k='-CHOOSE-', disabled=True)]
            ])

    layout = [
        [sg.VPush()],
        [column_left, column_right],
        [sg.VPush()],
    ]

    return gt.gui_window_with_graph('Known Temperature-Voltage Sensor
Calibration Equation', layout,
                                    gt.FIG_SIZE_WIDTH, gt.FIG_SIZE_HEIGHT,
False)

def temp_volt_calibrate_window_behavior(niDAQ, window, fig, figure_canvas_agg
):
    """
    Behaviour for temperature-voltage relation window
    :param niDAQ: object
    :param window: pysimplegui window
    :param fig: calibration plot
    :param figure_canvas_agg: canvas for the calibration plot
    :return: object with set calibration
    """
    calibration = ct.LinearCalibration('LEAST_SQUARES')
    while True:
        event, values = window.read()
        if event == sg.WINDOW_CLOSED:
            break

        if event == '-CHOOSE-':
            window.close()
            return calibration

        if event == '-LINEAR_EQ-':
            gt.set_disabled(window, False, '-LEAST_SQUARES-', '-
LINEAR_INTERPOLATION-')
            if not calibration.is_type('LINEAR_EQUATION'):
                calibration = calibration.to_linear_calibration()

        if event == '-CHOOSE_POINTS-':

calibration.set_chosen_points(select_points_window(calibration.get_data()))
```

```
        if event == '-NON_LINEAR_EQ-':
            gt.set_disabled(window, True, '-LEAST_SQUARES-', '-
LINEAR_INTERPOLATION-')
            gt.set_visible(window, False, '-CHOOSE_POINTS-')
            if not calibration.is_type('NON_LINEAR_EQUATION'):
                calibration = calibration.to_nonlinear_calibration()
            elif window['-EQ_EXPRESSION-'].metadata:
                # To calculate a nonlinear function there must be at least 3
points
                window['-EQ_EXPRESSION-'].update("Waiting for 3 points...")

        # only accepts digits and decimal point '.'

        if event in ['-V_INPUT-', '-T_INPUT-']:
            gt.filter_numeric_characters(window, values, event,
text_input_keys)

        if event == '-TOGGLE-':
            gt.gui_toggle_behaviour(window)

        if event == '-ENTER-':
            try:
                if values['-T_INPUT-'] == "":
                    raise ValueError("Values must be assigned")
                elif not gt.is_number(values['-T_INPUT-']):
                    raise ValueError("Values must be a numeric value.")

                if not window['-TOGGLE-'].metadata:
                    if values['-V_INPUT-'] == "":
                        raise ValueError("Values must be assigned")
                    elif not gt.is_number(values['-V_INPUT-']):
                        raise ValueError("Values must be a numeric value.")
                    inputValues = [float(values['-V_INPUT-']), float(values['-
T_INPUT-'])]
                else:
                    inputValues = [niDAQ.read_voltage(), float(values['-
T_INPUT-'])]

                if calibration.data_exists(inputValues):
                    raise ValueError("Data input is repeated.")

                calibration.add_data(inputValues)

            except ValueError as e:
                sg.popup_error(str(e), title="Error")

            window['-TABLE-'].update(values=calibration.data)
            window['-N_SAMPLES-'].update(len(calibration))
            window['-V_INPUT-'].update('')
            window['-T_INPUT-'].update('')

        if event == '-DELETE-':
            del calibration[-1]
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=False)

        if event == '-CLEAR-':
            calibration.clear_data()
            calibration.change_in_data(window, fig, figure_canvas_agg,
known_expression=False)
```

```
        if event == '-COPY-':
            window['-COPY-'].update('Text Copied!', disabled=True)
            sg.clipboard_set(repr(calibration))  # Copy the text to clipboard

            time.sleep(1)
            window['-COPY-'].update('Copy', disabled=False)

        if len(calibration) > 0:
            gt.set_disabled(window, False, '-CLEAR-', '-DELETE-')
            if values['-LINEAR_EQ-']:
                if len(calibration) > 1:
                    if values['-LEAST_SQUARES-']:
                        calibration.update_method('LEAST_SQUARES')
                        calibration.calculate_expression()
                    else:
                        calibration.update_method('LINEAR_INTERPOLATION')
                        if calibration.interpolation_points:

calibration.calculate_expression(calibration.interpolation_points[0],

calibration.interpolation_points[1])
                        else:

calibration.calculate_expression([calibration.sort_x()[0],
calibration.sort_y()[0]],

[calibration.sort_x()[-1], calibration.sort_y()[-1]])

                    window['-EQ_EXPRESSION-'].update(repr(calibration))
                    window['-EQ_EXPRESSION-'].metadata = True
                else:
                    # To calculate a polynomial function there must be at
least 2 points
                    window['-EQ_EXPRESSION-'].metadata = False
            elif values['-NON_LINEAR_EQ-']:
                if len(calibration) > 2:
                    calibration.calculate_expression()
                    window['-EQ_EXPRESSION-'].update(repr(calibration))
                    window['-EQ_EXPRESSION-'].metadata = True
                else:
                    window['-EQ_EXPRESSION-'].metadata = False
            calibration.update_figure(fig, figure_canvas_agg,
known_expression=False)
        else:
            gt.set_disabled(window, True, '-CLEAR-', '-DELETE-')
            gt.set_visible(window, False, '-COPY-')
            window['-EQ_EXPRESSION-'].metadata = False

        if window['-EQ_EXPRESSION-'].metadata:
            gt.set_disabled(window, False, '-CHOOSE-')
            gt.set_visible(window, True, '-COPY-')
        else:
            gt.set_disabled(window, True, '-CHOOSE-')
            gt.set_visible(window, False, '-COPY-')
            # To calculate a linear function there must be at least 2 points

            window['-EQ_EXPRESSION-'].update(f"Waiting for "
                                             f"{'2' if calibration.is_type(
'LINEAR_EQUATION') else '3'} points...")
```

```
        gt.set_visible(window, values['-LINEAR_INTERPOLATION-'] and values['-
LINEAR_EQ-'] and
                       window['-EQ_EXPRESSION-'].metadata, '-CHOOSE_POINTS-')

    window.close()
```

src/guiTools.py

```python
import matplotlib
import time
import re

import PySimpleGUI as sg

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from pathlib import Path

# ---------- APPEARANCE ----------
WINDOW_WIDTH = 900
WINDOW_HEIGHT = 600

FIG_SIZE_WIDTH = 5
FIG_SIZE_HEIGHT = 4

SIZE_INPUT = (7, 1)

DEFAULT_THEME = "GreenMono"
ACCENT_THEME = "LightGreen10"

FONT_DEFAULT = ('Helvetica', 10)
FONT_BOLD = ('Helvetica', 10, 'bold')

# ---------- TEXTS FOR WINDOWS ----------

# Calibration Method
CAL_METHOD_TEMP_VOLT = "Temperature and Voltage Relation"

CAL_METHOD_EXP = "Direct Expression Input"

# Temperature and Voltage Correlation
TEMP_VOLT_LIN_EQ = "Linear Equation"
TEMP_VOLT_LEAST_SQUARES = "Least Squares Method"
TEMP_VOLT_LIN_INTERP = "Linear Interpolation"
TEMP_VOLT_NON_LINEAR_EQ = "Non-linear Equation"

# Direct Expression Input
INP_EXP_LIN_EQ = "Linear Equation",
INP_EXP_NON_LIN = "Non-linear Equation"

# Data Acquisition
DATA_ON_DEMAND = "On Demand"
DATA_CUSTOM = "Finite Sampling"

# -------- IMAGE PATHS ---------
ICON_PATH = 'assets/icon_big.png'
TOGGLE_ON_PATH = 'assets/switch_on.png'
TOGGLE_OFF_PATH = 'assets/switch_off.png'
ALARM_MIN_ON_PATH = 'assets/alarm_min_on.png'
ALARM_MIN_OFF_PATH = 'assets/alarm_min_off.png'
ALARM_MAX_ON_PATH = 'assets/alarm_max_on.png'
ALARM_MAX_OFF_PATH = 'assets/alarm_max_off.png'
ALARM_UNSET_PATH = 'assets/alarm_unset.png'

# -------- PARAMETERS ---------
N_DECIMALS = 3
```

```
MIN_TIME_UPDATE_MS = 60  # minimum time that app can update

MAX_TIME_INTERVAL_MS = 5100


def get_desktop_dir():
    """
    Get the user's desktop directory
    :return: user's desktop as a string
    """

    return str(Path.home().joinpath("Desktop")) + "/"


def filter_digits(window, values, event, text_input_keys: list):
    """
    Filters out non-digit text inputs even if the user types letter, numbers
or symbols
    :param window: window from gui where text is inputted and shown
    :param values: list of values in gui window
    :param text_input_keys: list with text-input keys
    :param event: event in gui window
    :return:
    """
    k_event = text_input_keys[text_input_keys.index(event)] if len
(text_input_keys) > 1 else text_input_keys[0]
    values[k_event] = "".join(c for c in values[k_event] if c.isdigit())
    window[k_event].update(values[k_event])

def filter_numeric_characters(window, values, event, text_input_keys: list):
    """
    Filters out non-numeric text inputs so that even if the user types
letters and numbers, only numbers, '.' and '-'
    are shown
    :param window: window from gui where text is inputted and shown
    :param values: list of values in gui window
    :param text_input_keys: list with text-input keys
    :param event: event in gui window
    """
    # assigns text input key where there is an event

    k_event = text_input_keys[text_input_keys.index(event)] if len
(text_input_keys) > 1 else text_input_keys[0]
    # empty string where filtered out characters will be added

    filtered_chars = []
    # flag to signal if a '.' has already been typed in

    dot_found = False
    for char in values[k_event]:
        # adds if char is between 0-9
        if char.isdigit():
            filtered_chars.append(char)
        # adds '.' if it's the first one found

        elif char == '.' and not dot_found:
            filtered_chars.append(char)
            dot_found = True
        # adds '-' if it's in the first position

        elif char == '-' and len(filtered_chars) == 0:
```

```
            filtered_chars.append(char)

    values[k_event] = ''.join(filtered_chars)
    window[k_event].update(values[k_event])

def _check_if_key(key_input):
    """
    Private method that checks if the value passed is a valid key
    :param key_input:
    :return:
    """
    # Checks if value is a string
    if not isinstance(key_input, str):
        raise TypeError(f"Variable 'key_input' must be a string,\ngot {type
(key_input).__name__} instead.")
    # Checks if the value passed has the correct key format

    key_format = r'^-\w+-$'  # defining pattern: '-<key>-'

    if not re.match(key_format, key_input):
        raise ValueError(f"Invalid key format for 'key_input': '{key_input}'.
\nKeys must have '-<key>-'")

def check_if_valid_input(values, n_decimals, *args):
    """
    Checks if input value when entered is a valid number and isn't empty
    :param n_decimals: number of decimals desired
    :param values: list of values in gui window
    :param args: input key(s)
    :return: list of valid input(s) as floats with 3 decimals
    """
    # list where valid inputs will be stored

    valid_inputs = []
    for key_input in args:
        # checks if key input is valid
        _check_if_key(key_input)
        # checks if input value isn't empty
        if values[key_input] == "":
            raise ValueError("Values must be assigned")
        # checks if input is a valid number
        elif not is_number(values[key_input]):
            raise ValueError("Values must be a numeric value.")
        # converts input from string to float with 3 decimals and adds it to
the valid input list
        valid_inputs += to_number_n_dec(n_decimals, values[key_input])
    return valid_inputs

def set_disabled(window, is_disabled: bool, *args):
    """
    Updates disabled parameter of an element
    :param window: gui window
    :param is_disabled: bool, True: if element should be disabled, False if
not
    :param args: element key(s)
    :return:
    """
    for key_input in args:
        _check_if_key(key_input)
        window[key_input].update(disabled=is_disabled)
```

```
def set_visible(window, is_visible: bool, *args):
    """
    Updates visibility parameter of an element
    :param window: gui window
    :param is_visible: bool, True: if element should be visible, False if not
    :param args: element key(s)
    :return:
    """
    for key_input in args:
        _check_if_key(key_input)
        window[key_input].update(visible=is_visible)

def empty_inputs(window, *args):
    """
    Empties the input in a window
    :param window: gui window
    :param args: input key(s)
    :return:
    """
    for key_input in args:
        _check_if_key(key_input)
        window[key_input].update('')

def is_number(string):
    """
    Same as isnumeric but with floats also
    :param string: number input
    :return: True if it's a number, False if it isn't
    """
    if string[0] == '-':
        string = string[1:]

    return string.replace('.', '', 1).isdigit() or string.isnumeric()

def to_number_n_dec(n_decimals, *args):
    """
    Turns arguments to a float with 3 decimal points
    :param n_decimals: number of decimals desired
    :param args:
    :return:
    """
    if not isinstance(n_decimals, int):
        raise TypeError(f"Number of decimals must be integer,\ngot {type
(n_decimals).__name__}")
    result = []
    for number in args:
        if n_decimals > 0:
            result.append(round(float(number), n_decimals))
        else:
            result.append(round(int(number), n_decimals))
    return result

def calculate_frequency(period):
    """
    Calculates frequency value, given period value
    :param period: period value
    :return: frequency value in the same time unit as period value
    """
    if not isinstance(period, (int, float)):
        raise TypeError(f"Expected a number (float or integer), got {type
(period).__name__} instead")
```

```
    return 1 / period

def gui_toggle_behaviour(window):
    window['-TOGGLE-'].metadata = not window['-TOGGLE-'].metadata
    if window['-TOGGLE-'].metadata:
        set_disabled(window, True, '-V_INPUT-')
        window['-V_INPUT-'].update("")
        window['-V_TXT-'].update(text_color=sg.theme_button_color()[1])
        window['-TOGGLE_OFF_TXT-'].update(font=FONT_DEFAULT)
        window['-TOGGLE_ON_TXT-'].update(font=FONT_BOLD)
    else:
        set_disabled(window, False, '-V_INPUT-')
        window['-V_TXT-'].update(text_color=sg.theme_text_color())
        window['-TOGGLE_ON_TXT-'].update(font=FONT_DEFAULT)
        window['-TOGGLE_OFF_TXT-'].update(font=FONT_BOLD)

    window['-TOGGLE-'].update(image_filename=TOGGLE_ON_PATH if window['-
TOGGLE-'].metadata else TOGGLE_OFF_PATH)

def gui_window_with_graph(title, layout, figSizeWidth, figSizeHeight, isModal
):
    """
    Initializes a PySimpleGUI window with a matplotlib using a CANVAS with
empty graph that can be updated later
    :param title: title of the window
    :param layout: layout designed for the window
    :param figSizeWidth: desired width of the graph
    :param figSizeHeight: desired height of the grap
    :param isModal: bool if window is modal
    :return: window, fig, figure_canvas_agg
    """
    # Create the PySimpleGUI window with the provided title and layout

    window = sg.Window(title, layout, finalize=True, element_justification=
'center', modal=isModal,
                       size=(WINDOW_WIDTH, WINDOW_HEIGHT))
    # Create a new matplotlib Figure object with the provided size

    fig = matplotlib.figure.Figure(figsize=(figSizeWidth, figSizeHeight))
    # Adjust the position of the axes within the figure

    fig.subplots_adjust(top=0.8, bottom=0.25, left=0.2)  # Move the axes up
by adjusting the top and bottom positions

    # Add a subplot (axes) to the figure and plot an empty line

    fig.add_subplot(111).plot([], [])
    # Create a FigureCanvasTkAgg object by associating the figure with the
tkinter canvas element
    figure_canvas_agg = FigureCanvasTkAgg(fig, window['-CANVAS-'].TKCanvas)
    # Draw the initial empty plot on the canvas

    figure_canvas_agg.draw()
    # Pack the canvas widget into the window's layout

    figure_canvas_agg.get_tk_widget().pack()

    return window, fig, figure_canvas_agg

def get_axes_for_points(fig, data: list):
    """
```

```
    Creates axes for a graph made with data points
    :param fig:
    :param data: list of points
    :return: axes with xlabel, ylabel and a grid and x and y list of
separated data
    """
    axes = fig.axes
    x = [i[0] for i in data]
    y = [i[1] for i in data]
    axes[0].clear()
    axes[0].set_xlabel("Voltage (V)")
    axes[0].set_ylabel("Temperature (ºC)")
    axes[0].grid()
    return axes, x, y

def pack_canvas(figure_canvas_agg):
    """
    Packs canvas for later use in graph
    :param figure_canvas_agg:
    :return:
    """
    figure_canvas_agg.draw()
    figure_canvas_agg.get_tk_widget().pack()

def draw_points(axes, x_points, y_points, marker, label=None):
    """
    If a user selects a points in table, it will be drawn in table
    :param axes:
    :param x_points: x point or list
    :param y_points: y point or list
    :param marker: desired shape and color for point/s
    :param label: name of label
    :return:
    """
    axes[0].plot(x_points, y_points, marker, label=label)

def sort_pair_list_by_x(data):
    """
    Sorts a list of pairs by x
    :param data: list of data points
    :return: list of pairs sorted
    """
    sorted_points = sorted(data, key=lambda p: p[0])
    return sorted_points

def get_sorted_nth_elements(data, n):
    """
    Extracts elements at any index n from the pairs
    :param data: list of pairs
    :param n: nth element [0] or [1]
    :return: list of nth elements
    """
    return [i[n] for i in sort_pair_list_by_x(data)]
```