UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CAMPUS DO VALE - CAMPUS CENTRO

Samuel Huff Dieterich

DEVELOPMENT OF A CONTROL AND MONITORING SYSTEM FOR CRYOSTAT

PORTO ALEGRE
2024

SAMUEL HUFF DIETERICH

DEVELOPMENT OF A CONTROL AND MONITORING SYSTEM FOR CRYOSTAT

Final Course Project submitted to the Universidade Federal do Rio Grande do Sul, as a necessary requirement to obtain the Bachelor's degree in Engineering Physics

Porto Alegre, August 2024

Abstract

This project aims to develop a software for control and monitoring system for a cryostat, using the Cryomagnetics C-Mag Vari-9 Cryogen-Free as a reference located in Laboratório de Resistividade, Magnetismo e Supercondutividade (LabRMS) at IF-UFRGS. As basic goals, the software is intended to be able to control and monitor the temperature and magnetic field of the cryostat, according to pre-defined sequences of operations, and store the collected data for posterior analysis. For additional objectives, it is intended to develop a modular application that can be easily adapted to other equipment and have its functionalities expanded. Throughout this document, the engineering project will be presented, including the problem statement, the technical description of the equipment, the proposed solution, and the results achieved by this work. The main persona interested in the development of this work is LabRMS of IF-UFRGS, which has the pose of the cryostat that will be used for the development of this system. The laboratory is coordinated by Professor Milton Tumelero, who is also the advisor of this work. Furthermore, students and researchers inside and outside LabRMS can also benefit from the developed program, since, by design, it can be adapted to other equipment and consequently to other laboratories.

Keywords: Control and Monitoring System, Cryostat, Automation.

Resumo

O objetivo do presente projeto consiste em desenvolver um software para controle e monitoramento de um criostato, utilizando o Cryomagnetics C-Mag Vari-9 Cryogen-Free como referência, localizado no Laboratório de Resistividade, Magnetismo e Supercondutividade (LabRMS) do IF-UFRGS. Como objetivos básicos, o software deve ser capaz de controlar e monitorar a temperatura e o campo magnético do criostato, de acordo com sequências de operações pré-definidas, e armazenar os dados coletados para análise posterior. Como objetivos adicionais, pretende-se desenvolver uma aplicação modular que possa ser facilmente adaptada a outros equipamentos e ter suas funcionalidades expandidas. Ao longo deste documento, o projeto de engenharia será apresentado, incluindo a declaração do problema, a descrição técnica do equipamento, a solução proposta e os resultados atingidos por este trabalho. A principal persona interessada no desenvolvimento deste trabalho é o LabRMS da IF-UFRGS, que possui o criostato que será utilizado para o desenvolvimento deste sistema. O laboratório é coordenado pelo professor Milton Tumelero, que também é o orientador deste trabalho. Além disso, estudantes e pesquisadores dentro e fora do LabRMS também podem se beneficiar do programa desenvolvido, uma vez que, pelo projeto, pode ser adaptado a outros equipamentos e consequentemente a outros laboratórios.

Palavras-chave: Sistema de Controle e Monitoramento, Criostato, Automação.

List of Figures

Figure 1 – C-Mag Vari-9 Cryogen-Free Cryostat - Cryomagnetics, Inc. - in the Laboratório de Resistividade, Magnetismo e Supercondutividade (LabRMS). 16
Figure 2 – Tower with peripherals attached to the cryostat - on the left - and the computer - on the right - and other general purpose equipment. 18
Figure 3 – Temperature Monitor TM612 18
Figure 4 – A playful wireframe representation of the C-Mag cryostat main components based on the system drawing shown in the manufacturer manual. 19
Figure 5 – Temperature Controller Model 24C 19
Figure 6 – 4G Superconducting Magnet Power Supply 20
Figure 7 – Software architecture diagram. 21

List of Tables

Table 1 – Popular data serialization file formats. From left to right: JSON, XML, and YAML. 22
Table 2 – Comparison between the existing LabView program and the developed solution. 53

List of Acronyms

CSV Comma-Separated Values

GHS Gas Handling System

GPIB General Purpose Interface Bus

IF Instituto de Física

IP Internet Protocol

LabRMS Laboratório de Resistividade, Magnetismo e Supercondutividade

MVP Minimum Viable Product

SCPI Standard Commands for Programmable Instruments

UFRGS Universidade Federal do Rio Grande do Sul

USB Universal Serial Bus

VTI Variable Temperature Insert

TCP Transmission Control Protocol

Contents

1 INTRODUCTION . . . 11
1.1 Motivation . . . 11
1.2 Goals . . . 11
1.2.1 General Goal . . . 11
1.2.2 Goals . . . 12
1.3 Interested Parties . . . 12
1.4 Scope . . . 12
1.5 Organization . . . 13

2 SPECIFICATION DEVELOPMENT . . . 14
2.1 Usage Context . . . 14
2.2 Comparative Analysis . . . 14
2.3 Requirements . . . 15

3 TECHNICAL CHARACTERISTICS . . . 16
3.1 Cryostat . . . 16
3.1.1 Superconducting Magnet . . . 17
3.1.2 Cryostat Operation . . . 17
3.2 Peripherals . . . 17
3.2.1 Temperature Monitor . . . 17
3.2.2 Temperature Controller . . . 18
3.2.3 Magnet Field Controller . . . 20
3.3 Communication Protocols . . . 20

4 PROPOSED SOLUTION . . . 21
4.1 Input . . . 22
4.1.1 File Format . . . 22
4.1.2 File Provider . . . 22
4.1.2.1 System . . . 23
4.1.2.2 User . . . 23
4.1.3 Types of Input Files . . . 23
4.2 Parser . . . 24
4.2.1 Serialization and Deserialization . . . 24
4.2.2 Objects . . . 24
4.3 Execute . . . 25
4.3.1 Main Program . . . 25

4.3.1.1 Communication Protocols 25
4.3.2 Background Functions 26
4.3.2.1 Logging 26
4.3.2.2 Results 26
4.3.2.3 Real-Time Charts 27

5 DETAILED PROJECT 28
5.1 Tech stack 28
5.1.1 Programming Language 28
5.1.1.1 Python 29
5.1.1.2 C/C++ 29
5.1.1.3 Java 29
5.1.1.4 JavaScript/TypeScript 29
5.1.1.5 Rust 30
5.1.2 Tools 30
5.1.2.1 Rustup 30
5.1.2.2 Cargo 30
5.1.2.3 MinGW 30
5.1.2.4 Nix 31
5.1.2.5 Git 31
5.1.2.6 GitHub 31
5.1.3 Libraries 31
5.1.3.1 Serde 31
5.1.3.2 Tokio 31
5.1.3.3 Tracing 32
5.1.3.4 Tera 32
5.1.3.5 Regex 32
5.1.3.6 Clap 32
5.2 Implementation 32
5.2.1 Input files 33
5.2.1.1 Instructions 33
5.2.1.2 Devices 35
5.2.1.3 Pipeline 36
5.2.1.3.1 Instruction 39
5.2.1.3.2 Wait For 39
5.2.1.3.3 Scan 40
5.2.2 Deserialization 41
5.2.3 Communication 43
5.2.4 Instruction 44
5.2.5 Wait For 45

5.2.6 Scan 46

6 RESULTS AND CONCLUSIONS 48

6.1 Results 48

6.2 Requirements Evaluation 50

6.3 Future Work 51

6.3.1 Measures Block 51

6.3.2 Scoped Variables and Attributes 51

6.3.3 Configuration File 52

6.3.4 Meta Instructions 52

6.3.5 Graphical Interface 52

6.4 Highlights 53

References 55

1 Introduction

1.1 Motivation

The use of complex instrumental systems is a necessity both in industry and academia. While the industry requires measurement and control instruments to ensure the specified quality of its products, academia needs top-notch equipment to conduct research and develop new technologies. In this context, a system - as we will refer to it here - is formed by the combination of hardware and software. The hardware represents the physical part of the system, including electronic components, sensors, actuators, and casing, among others. On the other hand, the software represents the logical part of the system, in other words, the program that controls the hardware, performing predetermined functions and interacting directly with the user.

Although good hardware is often necessary for the success of a technology, the software sometimes is what sells the product. This is because the physical equipment is not always what brings the greatest added value, especially when the embedded software is essential for its use. For example, a temperature sensor can be built with relatively simple and inexpensive materials, such as a thermocouple and a signal conditioning circuit. However, the added value to the final product is mainly due to the software that calibrates the sensor and presents the temperature value in a user-friendly interface.

For more robust and specific instruments, manufacturers can often sell the hardware and software separately. In this case, the software is sold as a usage license, which can be acquired for a fixed price or through a monthly subscription. The cost of the license can be quite high, especially for research instruments where the number of users is low and the equipment cost is high. Moreover, the software may be proprietary, meaning that the user does not have access to the source code and cannot make modifications to it. This can be a problem for the user who may require functionality that is not present in the software or who wants to make modifications to improve its usability.

1.2 Goals

1.2.1 General Goal

Given the context presented in section 1.1, the general goal of this work is to develop a software for control and monitoring system for a cryostat, using the Cryomagnetics C-Mag Vari-9 Cryogen-Free as a reference located in Laboratório de Resistividade, Magnetismo e Supercondutividade (LabRMS) at IF-UFRGS.

1.2.2 Goals

- Develop a communication method between the computer and the cryostat;
- Control the temperature and magnetic field of the cryostat through the software;
- Monitor the temperature and magnetic field of the cryostat storing the data for posterior analysis;
- Write the software in a modular fashion, allowing it to be easily adapted to other equipment and have its functionalities expanded.

With these goals, the software will be capable of controlling and monitoring the cryostat, with minimum to no human intervention. For this manner, the system will be able to track the temperature and magnetic field of the cryostat, storing these data for subsequent analysis. Finally, the code will be modular, that way, it will be able to be used for different purposes even after the end of this project.

1.3 Interested Parties

The main persona interested in the development of this work is LabRMS of IF-UFRGS, which has the pose of the cryostat that will be used for the development of this system. The laboratory is coordinated by Professor Milton Tumelero, who is also the advisor of this work. Furthermore, students and researchers inside and outside LabRMS can also benefit from the developed program, since the idea is that it can be adapted to other equipment and consequently to other laboratories.

1.4 Scope

For this work, it is expected to develop a control and monitoring software using the C-Mag Vari-9 cryostat present in LabRMS as a reference. To develop this interface, programs or libraries will be used that can help speed up the implementation of this work, but it is also expected that a specific code will be developed for this work. Furthermore, as explained in the section 1.2, the program must be able to measure and control the temperature and magnetic field of the equipment.

It is not part of the objectives of this work, despite being a future possibility, to develop an interface for any other equipment. Furthermore, the development of functions that are not essential for the cryostat in question will not be prioritized, for example, communication with other communication protocols that are not used for the equipment model and peripherals presented.

1.5 Organization

The presented work is organized as follows:

Chapter 2: Problem statement, presenting the context of this project, its goals, and the system specifications.

Chapter 3: Technical description, presenting the main components of the cryostat equipment and the used communication protocols.

Chapter 4: Proposed solution, describing how the software is planned to be developed and how its main functions should work.

Chapter 5: Detailed project, showcasing how the software was developed, listing what goals were met.

Chapter 6: Conclusion, exposing the obtained results and suggesting pieces of advice for future works.

2 Specification Development

2.1 Usage Context

For the specific case of the studied cryostat, there is no official software made by the manufacturer to monitor or control the equipment. To solve this problem, Professor Milton Tumelero has developed a program using LabView that is capable of running some experiments automatically. However, he notes that the program is not only limited but lacks flexibility. In order to perform different tests, he finds himself modifying a significant portion of the code, which proves far from ideal. Moreover, the program in question is not capable of running multiple, sequential, experiments. That means that the user has to manually start each experiment, even the ones that involve the same study material. This inadequacy was the main motivation for the development of this work.

Within the laboratory community, most of the self-made programs are developed using LabView or C. C is a powerful and flexible programming language, but it is not easy to use and it is not very intuitive, especially for those who are not familiar with programming. LabView, on the other hand, is more intuitive and easy to use, since it is a graphical programming language, but it is not free of charge and it is not open-source.

In light of these limitations, the laboratory is receptive to exploring alternative solutions that are both flexible and easy to maintain, not only for this cryostat but also for other equipment. Professor Milton emphasizes that while a user-friendly interface is appreciated, the majority of users are adept at programming. Therefore, a well-documented program that effectively fulfills its purpose is already an interesting option.

2.2 Comparative Analysis

As discussed in section 2.1, the manufacturer of the specified cryostat does not provide any official software for the control or monitoring of the equipment. Furthermore, Professor Milton, this work advisor, is unaware of any alternative software that could fit the needs of this particular purpose.

In this context, asserting that there is no competition might be too categorical. To provide a comprehensive evaluation, we will compare the proposed work with the existing software utilized in the laboratory, which is LabView-based. LabView, being a graphical programming language, facilitates the control of various and occasionally intricate equipment. However, it is a proprietary and expensive software, which makes it less accessible to the general public. In addition to that, despite LabView's popularity in

laboratories and certain industries, it is conceivable that most students will not have any prior or subsequent contact with it after their academic pursuits.

Additionally, LabView, as a general-purpose application, addresses a wide array of issues but lacks specificity for the intended use case. Consequently, this work can capitalize on the opportunity to be developed from the ground up, catering specifically to the execution of experiment pipelines.

2.3 Requirements

Given that a cryostat's main functions are to control and monitor the temperature and magnetic field under specific conditions during an experiment, the following requirements were identified by the laboratory representative, Professor Milton Tumelero:

- **Read operations**: Also known as logging, this basic requirement refers to the ability to read some variables from the instrumentation equipment, such as temperature and magnetic field.
- **Set operations**: This requirement refers to two operations: `set temperature` and `set field`. The first one should define the temperature setpoint at the temperature controller, while the second one should set the magnetic field value at the magnet power supply.
- **Wait for**: This is a custom set of functions that allows the user to wait for a specific condition - that can be a time or a value (temperature or magnetic field) - to be reached before proceeding to the next step.
- **Scan**: This is a custom set of functions and it is the most complex requirement. Similar to a while or for routines, this function should scan a range of values (temperature or magnetic field) and run subroutines (measurements) during the scan according to specified conditions.
- **Run on a Windows machine**: The software should be able to run on a Windows machine, since the laboratory's computers are running this operating system and depend on it to run other software.
- **User-defined pipelines**: The software should allow the user to define a sequence of operations to be executed, no matter its size. This is a crucial requirement, as it allows the user to automate the process of running experiments and it is the main motivation for the development of this work.

3 Technical Characteristics

In this chapter, a brief description of the equipment and its instrumental peripherals is presented.

3.1 Cryostat

A cryostat is a device used to run a variety of physics and materials experiments at low temperatures (close to absolute zero Kelvin) and high magnetic fields. To do so, cryostats are equipped with a superconducting magnet, a vacuum chamber, and a temperature control system. The LabRMS is equipped with the C-Mag Vari-9 Cryogen-Free cryostat from Cryomagnetics, Inc., as shown in Figure 1.

Figure 1 – C-Mag Vari-9 Cryogen-Free Cryostat - Cryomagnetics, Inc. - in the Laboratório de Resistividade, Magnetismo e Supercondutividade (LabRMS).

3.1.1 Superconducting Magnet

The superconducting magnet is a solenoid made of twisted multi-filamentary NbTi wire in a copper matrix. According to the manufacturer, the magnet is designed for 90 kG (9.0 T) at helium 4.2 K temperature. Further details about the magnet's specifications can be found in the manufacturer's operating instructions manual (1).

3.1.2 Cryostat Operation

A cryogen-free cryostat is a design that does not require liquid cryogens to cool the equipment. However, the system uses helium gas in the Gas Handling System (GHS) to cool the Variable Temperature Insert (VTI) and as an exchange gas inside the sample space. Therefore, both the GHS and the sample space must be filled with helium gas before the system is cooled.

Without the GHS operating, the VTI and sample will typically cool to between 190 and 110 K (if left long enough), only by using a pulse tube refrigeration cycle. To cool below this temperature, down to 1.8K, the GHS recirculation must be operating, and complex steps to cool the system depending on the temperature range are described in the manufacturer's operating instructions manual. With that and the sample heater, the sample temperature can be very well controlled from 1.8 K to 300 K (1).

3.2 Peripherals

The peripherals are instruments that are connected to the cryostat and are used to control or monitor the cryostat's physical parameters. The main peripherals are the temperature monitor, the temperature controller, and the magnet field controller, which are shown in Figure 2. Given that, the software planned to be developed must be able to communicate with this instrumentation.

3.2.1 Temperature Monitor

The temperature monitor is a device that precisely measures the temperature at some specific places inside the cryostat. The C-Mag Vari-9 Cryogen-Free cryostat in the LabRMS is equipped with the TM612 model, figure 3, which has 2 input channels and it supports a wide range of temperature sensors, such as Cernox™, Platinum RTD, thermocouples, and others (2).

The temperature monitor is connected to the computer via an Ethernet cable. The equipment offers some advanced monitor features but it can only read the temperature of the sensors and it does not have any control over the cryostat.

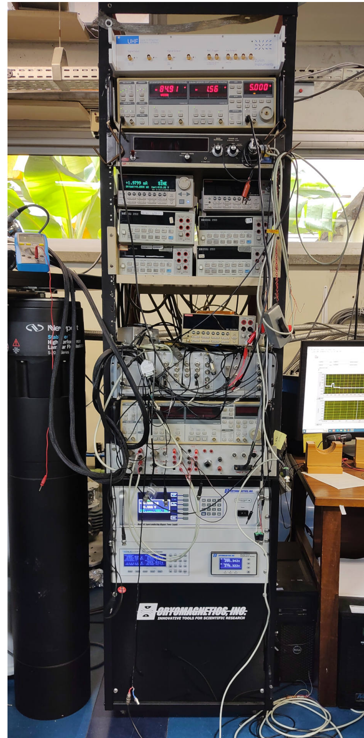

Figure 2 – Tower with peripherals attached to the cryostat - on the left - and the computer - on the right - and other general purpose equipment.

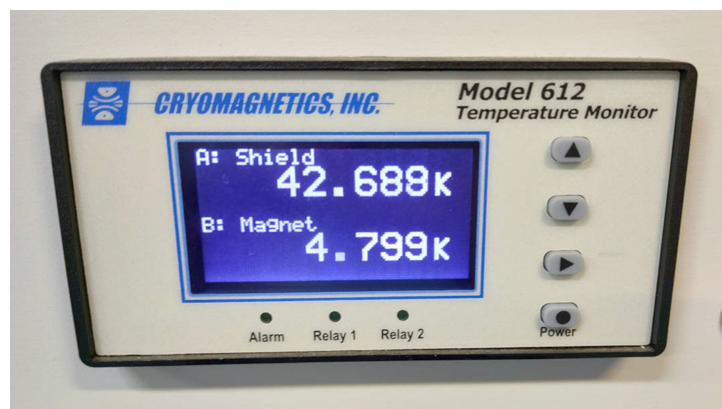

Figure 3 – Temperature Monitor TM612

3.2.2 Temperature Controller

The temperature controller is a device that not only measures the temperature of some points in the cryostat but also controls the temperature of the equipment during the experiment. The C-Mag Vari-9 Cryogen-Free cryostat in the LabRMS is equipped with the Model 24C, figure 5, which has 4 input channels and 2 control loops (3).

In Figure 4, the estimated positions of the temperature sensors are shown in red rings. Currently, there are 4 temperature sensors installed in the cryostat, with the names representing the location of the sensor: `Shield`, `Magnet`, `VTI`, and `Sample`.

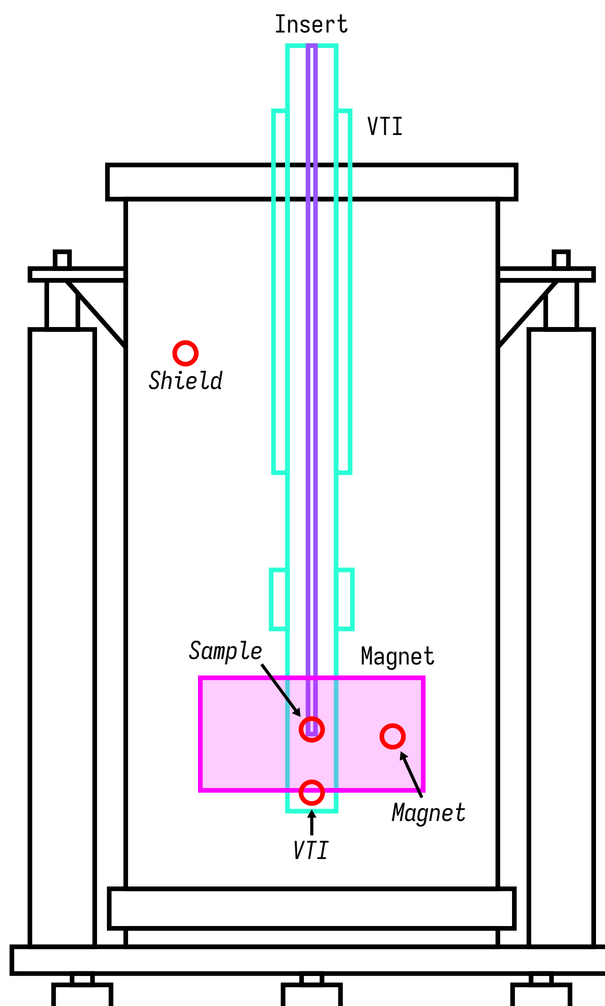

Figure 4 – A playful wireframe representation of the C-Mag cryostat main components based on the system drawing shown in the manufacturer manual.

In opposition to the temperature monitor, the temperature controller must be configured and receive commands and parameters from the user. The equipment is connected to the computer via an Ethernet cable where via remote control the user can set the temperature setpoint, the ramp rate, and the PID parameters.

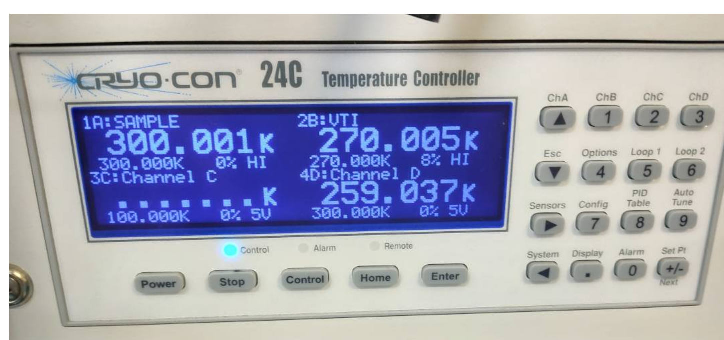

Figure 5 – Temperature Controller Model 24C

3.2.3 Magnet Field Controller

The magnet field controller is a device that controls the magnetic field generated by the superconducting magnet, since the current in the magnet is directly related to the magnetic field, the controller is a current power supply. The C-Mag Vari-9 Cryogen-Free cryostat in the LabRMS is equipped with the 4G Superconducting Magnet Power Supply, figure 6, which is a true four-quadrant power supply, meaning it is capable of operating sourcing and sinking power in both current and voltage (4).

The equipment is connected to the computer via an Ethernet cable where via remote control the user can set the current setpoint, which is directly related to the magnetic field, and the rate of change of the current.

Figure 6 – 4G Superconducting Magnet Power Supply

3.3 Communication Protocols

The main peripherals presented in the previous section can be controlled via TCP/IP, which is an easy and common communication protocol. However, some advanced instruments used in this equipment are limited to other options, such as GPIB and, in the future, Serial USB. Given that, for the scope of this work, the proposed solution must be capable of communicating with the devices via TCP/IP. Nonetheless, the software architecture must also be designed to be easily extensible to other communication protocols for future works.

4 Proposed Solution

The proposed solution consists of a software architecture that can be described in three phases: input, parser, and execute. The input phase is responsible for receiving the input files, instructions, and configurations, from the user. The parser phase is responsible for deserializing the input files into objects that can be used by the program. Lastly, the execute phase is responsible for executing the instructions and outputting the results.

To illustrate the proposed solution, figure 7 shows a diagram of the software architecture which will be explained in detail in the following sections.

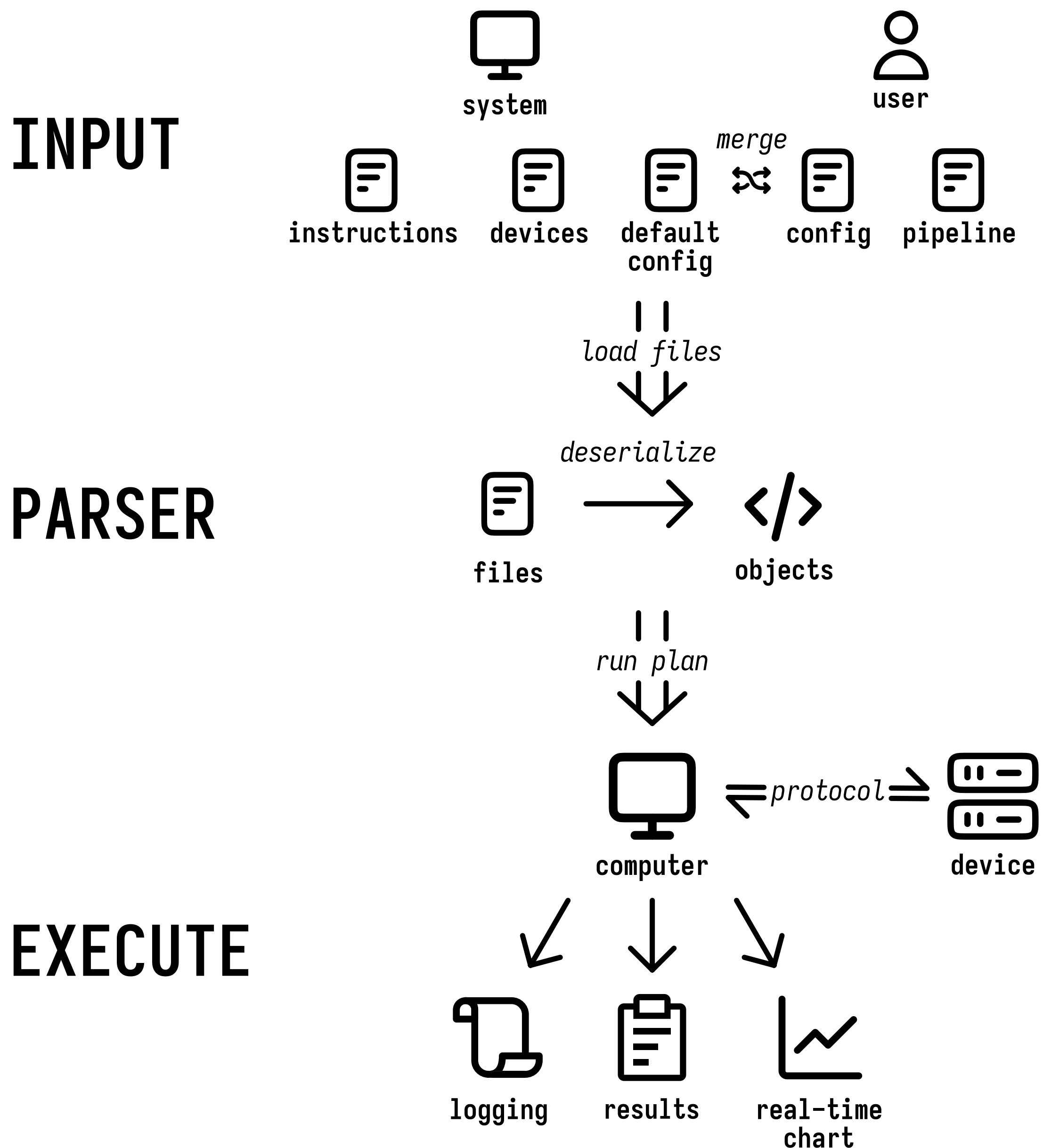

Figure 7 – Software architecture diagram.

4.1 Input

Most computer programs require some sort of input to work or to perform a specific task. In the case of the proposed program, the input data will be initially provided by the user through text files. Therefore, the first step of the architecture consists of reading the input files that will be further detailed in the following subsections. Before getting into the details of the types of input files, it is important to explain the format of these files as well as who will be responsible for making them available to the program.

4.1.1 File Format

The chosen input file format was YAML which is a human-readable data serialization language - a concept that will be better explained in section 4.2.1. In addition to being a popular format so virtually all programming languages can read it using a library, it is easier to read and write for humans as well, which can't be said for JSON and XML which are also popular alternatives. Those formats, as shown in the table 1, are more verbose and harder to manipulate and can impair the user experience considering that at first users will need to write the input files by hand.

JSON

```
{
  "users": [
    {
      "name": "John",
      "age": 20
    },
    {
      "name": "Mary",
      "age": 19
    }
  ]
}
```

XML

```
<users>
  <name>John</name>
  <age>20</age>
</users>
<users>
  <name>Mary</name>
  <age>19</age>
</users>
```

YAML

```
users:
  - name: John
    age: 20
  - name: Mary
    age: 19
```

Table 1 – Popular data serialization file formats. From left to right: JSON, XML, and YAML.

4.1.2 File Provider

As shown in figure 7, the input files can be provided by two different sources: the system and/or the user. The system - which can be the software itself or administrator users - will provide the default input files that are required for the program to work. The user, on the other hand, will provide the input files that are specific to the experiment that he wants to run.

4.1.2.1 System

The input files provided by the system are the instructions, devices, and the default configuration for the program. The instructions files contain the possible operations that a set of devices can perform. The devices file contains the details of the devices that are available to the program. Lastly, the default configuration file contains the default values for the configuration parameters that can be changed by the advanced users.

4.1.2.2 User

The input files provided by the user are the configuration and pipeline files. The configuration file is used to overwrite the default configuration provided by the system. The pipeline file describes the experiment by specifying the instructions that the program should execute and the devices that should be used to execute them.

4.1.3 Types of Input Files

The input files, as shown in figure 7, can be divided into four different types: instructions, devices, configuration, and pipeline.

- **Instructions**: Contain the instructions that different devices can execute. Since multiple devices can share the same instructions, the instructions can be separated from the devices to avoid redundancy. For instance, all devices that follow the SCPI standard can share the same instruction file containing the SCPI basic commands.
- **Devices**: Describe the devices that are available in the system. Each device must have an identification and a definition of how to communicate with it, as well as the instructions that it can execute. A device can also have some default configuration parameters that can be overwritten by the user.
- **Configuration**: Contains the configuration parameters that can be changed by the user. The configuration file is optional and can be used to overwrite the default configuration provided by the system.
- **Pipeline**: Defines the experiment steps by specifying the instructions that the program should execute and the devices that should be used to execute them. The pipeline file is mandatory and is used to describe the experiment that the user wants to run.

In chapter 5, the input files will be further detailed, showing their syntax and examples of how they can be set up.

4.2 Parser

The second phase of the software architecture is the parser. Generally speaking, a parser is a program that reads some text and converts it into a data structure that can be used by other programs. In the case of the proposed solution, the parser is responsible for reading the input files and converting them into their respective objects. In this step, it is possible to run some basic verifications to avoid runtime errors that could cause the program to crash.

4.2.1 Serialization and Deserialization

While calling this process a parser is not wrong, deserialization is a more accurate term. Deserialization is the process of converting a text format (string) into an object, more commonly used for object-oriented programs. On the other hand, serialization is the reverse process, converting an object into a text format. In the case of the proposed solution, as shown in figure 7, the parser - initially - will only perform the deserialization process. In other words, it is expected for the first version of the program to only read already formatted and structured input files written by the user as described in section 4.1. However, in the future, with the help of a user interface, the program could help the user to create those objects and then the serialization process would be equivalent to saving those configurations in the disk.

4.2.2 Objects

For each type of input file, there is a corresponding object that is used to store the information of that file. This is not only used to convert the YAML files to "binary data" to read its content but also to perform some functions inherited from the object.

Firstly, if it is not clear yet, there is a dependency between the objects. The pipeline object, for instance, depends directly on the device objects since they are defined in the input file to know which device is going to operate. Moreover, the device object has a strict dependency on the instruction objects since they will convert the instruction names into the actual commands that should be sent to the device. The configuration object is not as connected to the other objects because the idea is just to add some extra parameters that are not strictly limited to one object (experiment scoped).

With that in mind, we can understand the basic concept of object-oriented programming languages. Objects are not only supposed to store data but also to perform some methods or operations. Following the explanation of the previous paragraph, the device object should be able to send a command to the device using a specific protocol. The instruction object should be able to convert the instruction name into the actual command that should be sent to the device. More objects and methods will be defined in

the program development but this can give a basic overview on how the user-defined text will be used in the application.

4.3 Execute

The last phase of the software architecture is the execution of the instructions. This phase is responsible for executing the pipeline instruction sequentially by sending the commands to the devices and receiving their responses. Furthermore, given the provided output, the application should also be able to log the program execution, store the results in a data file, and potentially show a real-time chart with the selected physics attributes. These functionalities will be named background functions and will be explained in the following subsections. Nonetheless, the main program will be responsible for performing the experiment.

4.3.1 Main Program

After the input files are parsed, the main program will be responsible for running the plan that was defined by the user. To do so, it will iterate through the pipeline instructions and execute them sequentially in the specified device. The main program will also be responsible for handling the errors and exceptions that may occur during the execution of the instructions. Since this is a program that will control a scientific and potentially dangerous experiment, it is important to handle the errors and exceptions in a way that the program can recover from them and continue the execution or at least go to a safe state - which should be defined by the system (4.1.2.1). In order to perform any of those actions, the program must be capable of communicating with the devices and, therefore, the communication protocols will be explained in section 4.3.1.1.

4.3.1.1 Communication Protocols

The program can communicate with the devices using different communication protocols. The communication protocols are defined in the device object and, therefore, the main program should be able to communicate with the devices using the specified protocol.

Given the target goal of this project, the LabRMS cryostat, the communication protocols that are planned to be supported are TCP, GPIB, and Serial. The TCP protocol is the most common protocol used in the industry and it is used to communicate with devices that are connected to the network (wireless or network cable). The GPIB protocol is a communication protocol that is used to communicate with devices that are connected to a GPIB bus. Lastly, the Serial protocol is used to communicate with devices that are connected to the computer using a serial port which can be an USB, for instance. The

reason why we need to support multiple protocols is that the LabRMS cryostat uses different communication protocols for different devices. Although the TCP protocol is probably easier to use, some equipment only supports the GPIB protocol. In regards to the Serial protocol, there is a plan to upgrade the equipment with some new devices - such as an Arduino or a Raspberry Pi - that will be connected to the computer using a serial port.

Thankfully, for most programming languages, some libraries can be used to communicate with devices using those protocols. In addition to that, most devices - which includes the LabRMS cryostat peripherals - follow the Standard Commands for Programmable Instruments (SCPI) standard which defines a common syntax for the commands that should be sent to the devices. Therefore, although the program should have a generic query command that can be adapted to different protocols, the way that the read and write commands are sent to the devices should follow the SCPI pattern.

4.3.2 Background Functions

The background functions are functionalities that are not directly related to the execution of the instructions or should run in parallel with the execution of the instructions. That means that the background functions should not block the execution of the instructions, running in parallel or asynchronously with the main program all the time. The background functions that are planned to be implemented are: logging, results, and real-time charts.

4.3.2.1 Logging

Logging is a common practice in software development which is not only important during development to debug the program but also to keep track of the program execution during production. In the case of the proposed solution, the logging will be used for both purposes.

The logging will have two main streams: console and file. The console stream will be used to show the user the current state of the program execution and will be especially useful during the development of the program. The file, or multiple files, will be used to store the program execution instructions and states for future reference. That way, if the program crashes, the developer/user/administrator can check the log file to properly identify the error and fix it.

4.3.2.2 Results

The results background function is responsible for storing the results of the experiment in one or multiple data files. The data files will be used to store the results of the

scan instructions pre-defined by the user. Since different devices or measures can be used in the same scan, it is possible that during development it will be preferred to store the results in different files to avoid timing issues.

It is also possible that by design or user/administrator option, the program will store all the possible data during runtime and then filter the data that is necessary for the user in a separate process. That way, the user can roll back to the original and complete data file if necessary. However, knowing that the experiment can take a long time to finish and with a high-frequency collection of data, the data files can become very large and, therefore, use a lot of disk space per experiment.

It is possible that the program could support multiple output formats since different users may prefer different extensions and it should not be hard to implement. However, the default output format will most probably be CSV since it is a very common and easy-to-read and-write format.

4.3.2.3 Real-Time Charts

The program could have a graphical interface to improve the user experience. One of the features that could be implemented in the graphical interface is a real-time chart that could show the user the current state of the experiment. The way it will be implemented, if it is implemented since it is part of the additional goals, is not yet defined since it could range between some options depending on other factors. For instance, if by design the program stores all the data it can read all the time, the graph could plot basically any of the data that is being collected. On the other hand, the program could also be limited to only plotting what is being measured during a scan plus some basic information such as the temperature and magnetic field in the case of the cryostat. However, as a background function, this process should not block any other component of the application, including the operating system itself. So, it is important to do some tests and benchmarking to trade-off between the performance and the features that will be implemented.

5 Detailed Project

The software developed for this project is a command-line application that allows the user to communicate with different devices and collect data from them. The software was developed using the Rust programming language and it is capable of reading input files, communicating with devices, and executing pipelines. The following sections will discuss the software in more detail, including the chosen tech stack, the main components of the software, and the reasons they were developed that way. The main challenges and achievements will also be discussed, as well as ideas for future projects based on this one.

5.1 Tech stack

The tech stack is a set of tools, libraries, and programming languages that are used to develop a software project. The tech stack chosen for this project is based on the criteria defined in section 5.1.1. The following subsections will discuss the programming language, tools, and libraries that were used to develop the software and the reasons they were chosen.

5.1.1 Programming Language

Many different programming languages could be used to develop this application. Given that, it is necessary to choose one based on some criteria, such as:

- **Basic capabilities**: The language must be capable of performing the basic tasks required by the software. For example, it must be able to read and write files and communicate to devices through different protocols.
- **Performance**: Given that the application will mainly be used to collect as much data as possible, it needs to be able to handle heavy loads. Furthermore, the computer running the software will also be used for other tasks, so it is important that the software does not consume too much resources.
- **Development experience**: It is convenient to choose a language that most that attempt to improve the software are familiar with or can easily learn. This will make it easier to maintain and update the software in the future.
- **User experience**: This is not directly related to the programming language, but it is important to consider how the software will be installed and used. For example, it is important that the software can be easily installed on different operating systems and that it can be used without much technical knowledge.

- **Extra features**: In case the main goal of the project is achieved, other features could be added to the software. For example, a graphical interface could be added to make it easier to use.

Given these criteria, the following subsections will discuss some programming languages that could be used to develop the software and the reasons why they are or are not suitable for the project.

5.1.1.1 Python

Python is a great language for beginners and it is easy to use. It is also a great language for data analysis and manipulation, because of some popular math libraries such as NumPy and Pandas. However, Python is not the best language for performance since it is an interpreted language which adds some overhead. For the same reason, it is not the best language for developing desktop applications since it would require the user to have a Python interpreter installed. Given that, Python could be used to develop the software, but it is not the best language considering the long-term goals of the project.

5.1.1.2 C/C++

C and C++ are great languages for performance, but they are not the easiest languages to learn and use. They are also low-level languages, requiring the programmer to manage memory and other resources. Despite that, they are relatively popular - especially in academia - and they can be used to develop desktop applications for different operating systems. Some graphical libraries can be used to develop the software if we get to that point. Given that, C/C++ could be used to develop the software, but it could be hard to learn and use for most of the people who will attempt to improve the software.

5.1.1.3 Java

Java is a famous language but it is hard to find people that are willing to learn it nowadays. It is also not the best language for performance since it runs on a virtual machine. There are many different desktop applications developed in Java, showing that it is capable of doing so. However, especially because it is not being adopted as much as other languages, it was not considered a good option for the project.

5.1.1.4 JavaScript/TypeScript

JavaScript and TypeScript are dominant languages for web development and they are easy to learn and use. Although they could produce great graphical interfaces, they are not the best languages for performance and use in desktop applications. Some libraries can

help with that, but they basically run on top of a web browser which uses a lot of resources. Given that, JavaScript and TypeScript were disqualified as options for the project.

5.1.1.5 Rust

Rust is a great language for performance, comparable to C/C++, but it is also memory-safe. It is also a low-level language, which usually tends to be harder to learn but it is by design develop friendly. Because of that, its popularity and community have skyrocketed which explains its consecutive wins as the most-loved programming language according to the Stack Overflow Developer Survey (5). Therefore, even if it is relatively new and may not have as many developers as other languages, it is easy to convey that it is a great language to learn. Given that, Rust was chosen as the programming language for the project.

5.1.2 Tools

In addition to the programming language, some tools will be essential to develop the software. The following subsections will discuss some of these tools and the reasons they are necessary.

5.1.2.1 Rustup

Rustup is the recommended tool to install Rust. It is also used to manage different versions of the Rust compiler and to install different targets. By different targets, it means different operating systems and architectures. Given that, Rustup is essential to develop the software and to compile it for different machine combinations.

5.1.2.2 Cargo

Cargo is the package manager for Rust. It is used to create, build, test, and benchmark Rust projects. It is also used to manage dependencies and to publish the software, similar to what Pip is for Python. Given that, Cargo is an essential tool to develop complex libraries and applications in Rust.

5.1.2.3 MinGW

Speaking of different machine combinations, MinGW is a tool that allows compiling the software for Windows in a Unix-like environment. That way, it is possible to compile the software for Windows without using a Windows machine just to compile the software.

5.1.2.4 Nix

Nix is a package manager and programming language that allows the creation of reproducible builds. It has many features, such as the ability to create isolated environments, to manage different versions of the same package, and to create custom packages. With that, Nix can be used to create a development environment that is the same for all developers using a Unix-like operating system, capable of compiling the software for different machine combinations (including Windows).

5.1.2.5 Git

Git is a distributed version control system. It is used to track changes in the source code and to coordinate the work of different developers. That way, it is possible to maintain different versions and stages of the software, always being able to go back to a previous version if necessary. Git is the most popular tool for version control and it will be used to develop this project as well.

5.1.2.6 GitHub

GitHub is a web-based platform that uses Git for version control. It is also used to host the source code and much more, such as an issue tracker, releases, project milestones, automatic CI/CD pipelines, and more. GitHub is the most popular platform for open source projects and it will be used to host the source code of the software.

5.1.3 Libraries

It is expected that the software will be developed using some libraries to help with some tasks. The following subsections will discuss some of the main libraries that were used for this project and the reasons they are necessary.

5.1.3.1 Serde

Serde is a framework for serializing and deserializing Rust data structures. It is used to convert data from and to different formats, such as JSON, YAML, XML, and more. Given that, Serde is essential to read and write input files that were defined in section 4.1. Since the input files are defined in YAML, the Serde YAML library will be used for this project.

5.1.3.2 Tokio

Tokio is an asynchronous runtime for Rust. It is used to develop asynchronous applications, such as network servers and clients. That way, it is possible to communicate

with different devices and collect data from them without blocking the main thread. Given that, Tokio will be used to allow this asynchronous capabilities in the software.

5.1.3.3 Tracing

Tracing is a framework for instrumenting Rust programs with context-aware, structured, event-based diagnostic information. It is used to log information about the software execution, such as errors, warnings, and debug information. Given that, Tracing is essential to help with the development and maintenance of the software. It will also be essential for some background functions, such as the logging and results explained in sections 4.3.2.1 and 4.3.2.2.

5.1.3.4 Tera

Tera is a template engine for Rust. It is used to generate string and text outputs based on templates, using double curly braces (`{{}}`) to insert variables and curly braces with percentage signs (`{% %}`) to insert control structures. Given that, Tera is essential to format the command query and also to parse the response from the devices.

5.1.3.5 Regex

Regex is a regular expression library for Rust. It is used to match patterns in strings, such as numbers, words, and special characters. Given that, Regex is needed to parse the response from the devices, making it possible to extract the values from it following a pattern.

5.1.3.6 Clap

Clap is a command-line argument parser for Rust. It is used to parse the command-line arguments and options passed to the software. Given that, Clap is included to allow the user to configure the software and to run it with different options and input parameters.

5.2 Implementation

The software was developed using the Rust programming language and some libraries that were discussed in the previous section. The main components of the software were designed to allow the user to communicate and interact with different devices through a pipeline configuration containing 3 types of steps: instruction, wait for, and scan.

The following subsections will explain the main components of the system and the reasons they were designed that way. The code is available in the FlowLab GitHub

repository (https://github.com/SamuelHDieterich/flowlab). It is important to mention that the features and limitations explained here may not be up-to-date with the current version of the software, as it can be updated after the submission of this work.

5.2.1 Input files

The input files are used to define the devices, instructions, and pipelines that the software will use, as it was proposed in section 4.1. The configuration type that was explained in subsection 4.1.3 was not implemented for this project but some use cases were considered.

As it was explained in subsection 4.1.1, the input files are defined in YAML format, which is a human-readable data serialization standard. The following topics will explain the file syntax for each type of configuration and the reasons they were designed that way.

5.2.1.1 Instructions

```yaml
instructions:
  - name: InstructionName
    command:
      query: "{{command}} {{argument1}} {{argument2}} ..."
      parameters:
        - name: argument1
          type: string|integer|float|boolean
          default: <Default value> # Optional
          values:
            - <Value 1>
            - ...
          description: <Description of the argument> # Optional
        - ...
    response: # Optional
      format: "{{output1}} {{output2}} ..."
      parameters:
        - name: output1
          type: string|integer|float|boolean
          values: # Optional
            - <Value 1>
            - ...
          description: <Description of the output> # Optional
        - ...
```

```yaml
    description: <Description of the instruction> # Optional
  - ...
```

An instruction file contains a list of instruction objects. Each instruction object has a name, command, response (only needed for commands that expect a return), and description.

- `name`: The name of the instruction that should be describable and unique, usually specified by the device manufacturer manual.
- `command`: This block defines the command string and possible parameters that should be passed to the device to execute the instruction.
  - `query`: The command that should be sent to the device to execute the specified instruction. The parameters, if any, are specified between double curly brackets.
  - `parameters`: The list of parameters that the command expects.
    - `name`: The name of the parameter. This name must be used in the query field.
    - `type`: The type of the parameter. The possible types are: `string`, `integer`, `float`, and `boolean`.
    - `default`: The optional default value of the parameter.
    - `values`: If only a handful of values are allowed to a parameter, a list of possible values can be used in this situation.
    - `description`: An optional description of the parameter.
- `response`: If the command expects a return (read operation), the response object should be defined.
  - `format`: The format of the response. The output variables are specified between double curly brackets.
  - `parameters`: The list of output variables that the response should return.
    - `name`: The name of the output variable. This name must be used in the format field.
    - `type`: The type of the output variable. The possible types are: `string`, `integer`, `float`, and `boolean`.
    - `values`: If only a handful of values are allowed to an output variable, a list of possible values can be set so the user can know what to expect.
    - `description`: An optional description of the output variable.
- `description`: A description of the instruction.

The reason why the instruction file was designed to start with a "instructions" key instead of just the list of instructions is to allow the definition of other instruction objects in different places. This way, it is possible to define a list of instructions in different files and then merge them into a single list.

5.2.1.2 Devices

```yaml
devices:
  - name: DeviceName
    description: <Description of the device> # Optional
    protocol:
      # TCP protocol
      ip: 192.168.1.1
      port: 1234
    instructions:
      - path: InstructionFile1  # Recommended approach
      - name: <Instruction definition>
        ... # Rest of the instruction definition
    default_values: # Optional
      - name: <Parameter 1 value>
        value: <Parameter 1 value>
        description: <Description of the parameter>
  - ...
```

A device file contains a list of device objects. Each device object has a name, description, protocol, instructions, and default values (optional).

- `name`: The name of the device that must be unique.
- `protocol`: The communication protocol definition for the device. Depending on the protocol, different parameters are required.
  - `TCP`: TCP communication protocol. As shown in the example, the `ip` and `port` fields are required.
  - `GPIB`: GPIB communication protocol. It was not implemented in this project but it could be defined with the `address` field.
  - `Serial`: Serial communication protocol. It was not implemented in this project but it could be defined with a `port` and `baudrate` fields.
- `instructions`: The list of instructions that the device supports. It is possible to define the instructions in the same file or to reference an instruction file (recommended approach).

  - `path`: The file path of the instruction file that contains the instructions that the device supports.
  - `name`: This starts a new instruction definition that will be used only for this device. This is an alternative approach to defining the instructions and it follows the same pattern as described in the previous section 5.2.1.1.

- `default_values`: The default values that should be used when executing an instruction with this device. This is an optional field and it is useful when it is known that some parameters are always the same.
  - `name`: The name of the parameter.
  - `value`: The value of the parameter.
  - `description`: An optional description of the parameter.
- `description`: An optional description of the device.

In the same way as the instruction file, the device file was designed to start with a "devices" key instead of just the list of devices to allow the definition of other device objects in different places, such as different files, and then merge them into a single list.

It is important to mention that the same physical device can be defined multiple times with different names and configurations. For example, the TM612 temperature monitor has two read channels, so instead of defining a single device and setting the channels as parameters for each instruction, it is possible to define two devices with the same configuration (instructions and protocol) but different names and default values.

5.2.1.3 Pipeline

```yaml
name: PipelineName
description: <Description of the pipeline>

# Using the YAML anchor feature to reuse instructions
instructions:
  - &InstructionName
    name: <Instruction name>
    device: <Device name>
    parameters:
      - name: <Parameter name>
        value: <Parameter value>
      - ...
  - ...
```

```
# Define the devices to be used in the pipeline
devices:
  - path: <Device file path> # Recommended approach
  - name: <Device definition> # Alternative approach
    ... # Rest of the device definition
    instructions:
      # Add an instruction defined on the current file
      - <<: *InstructionName
  - ...

pipeline:
  # Simple instruction call
  - step: <Instruction name>
    device: <Device name>
    parameters:
      - name: <Parameter name>
        value: <Parameter value>
      - ...
  # Wait for a metric to reach a certain value or time
  - step: Wait for
    # Instruction to get the metric value
    # In case of a time condition, the metric field could be omitted
    metric:
      instruction: <Instruction name>
      device: <Device name>
      parameters: # Optional
        - name: <Parameter value>
          value: <Parameter value>
        - ...
    # Condition to continue the execution
    parameters:
      # This should be available in the metric response
      name: <Metric variable name>
      value: <Value to reach>
      tolerance: <Tolerance>
      # In case of a time condition,
      # only the delay field should be defined
      delay: <Delay in seconds>
  # Perform a scan through a range of values
```

```yaml
- instruction: Scan
  metrics:
    # List of metrics that will be performed
    # sequentially for each iteration
    - step: <Metric name>
      device: <Device name>
      parameters:
        # One of the parameters should be the variable,
        # so it doesn't need to be defined here
        # - name: <Parameter value>
        #   value: <Parameter value>
        - ...
    - ...
    # It is important to ALWAYS define a
    # Wait for step at the end of the scan.
    # Otherwise, the program will not wait
    # to reach the desired value
    - step: Wait for
      ... # Rest of the Wait for definition but one of
          # the parameters should be the scan variable
  type: settle|sweep
  parameters:
    variable: <Variable name>
    start: <Start value>
    stop: <Stop value>
    step: <Step value>
  # If not defined, the measures results will not be stored
  # The filepath can be a template string that will be
  # rendered with the available variables.
  # Example: data/{{PIPELINE_NAME}}_{{DATE}}_{{variable_name}}.csv
  datafile: <Datafile path>
  measures:
    # List of measures that will be performed for each iteration
    - step: <Measure name>
      device: <Device name>
      parameters:
        - name: <Parameter value>
          value: <Parameter value>
        - ...
```

```
    - ...
  - ...
```

The pipeline file defines a plan of instructions that should be executed in a specific order. It contains a name, description, instructions, devices, and the pipeline - a list of steps to perform. The name will be used to identify the pipeline during the execution and can also be used as a template string that will be explained in the scan step. The description, for now, is just a comment that can be used to describe the pipeline.

The instructions definition is optional and it can be used to define very specific instructions that will be used only in the scope of the pipeline. Furthermore, as shown in the example, it is possible to use the YAML anchor feature to reuse the defined instruction into multiple devices in the pipeline. However, the recommended approach is to define the instructions in the instruction file and reference them in the devices and pipeline files.

The devices key is used to define the devices that will be used in the pipeline. The cleaner and recommended approach is to reference a device file that contains the devices that will be used in the pipeline. However, it is also possible to define the devices in the pipeline file itself. This can be used for prototyping and to also use the instructions created in the same file using the YAML anchor feature.

Lastly, the pipeline key is used to define the steps that should be executed in the pipeline. The steps can be of three types: instruction, wait for, and scan. The instruction type is used to execute a single instruction, the wait for type is used to wait for a metric to reach a certain value or time, and the scan type is used to perform a scan through a range of values of a metric and measure different values for each step. The next paragraphs will break down each type of step.

5.2.1.3.1 Instruction

- `step`: The name of the instruction that should be executed.
- `device`: The name of the device that should execute the instruction.
- `parameters`: The list of parameters that should be passed to the instruction, if needed.
  - `name`: The name of the parameter.
  - `value`: The value of the parameter.

5.2.1.3.2 Wait For

- `step`: Since it's a built-in instruction, the name of the instruction is `Wait for` to invoke this special function.

- `metric`: The metric that should be monitored. In case of a time condition, the `metric` field could be omitted.
  - `instruction`: The name of the instruction that should be executed to get the metric value. Trivially, this instruction should have a response.
  - `device`: The name of the device that should execute the instruction.
  - `parameters`: The list of parameters that should be passed to the instruction, if needed.
    - `name`: The name of the parameter.
    - `value`: The value of the parameter.
- `condition`: The condition that should be met to continue the execution. In case of a time condition, only the `delay` field should be defined.
  - `name`: The name of the metric variable that should be monitored.
  - `value`: The value that the metric should reach.
  - `tolerance`: The tolerance of the metric value (`value` ± `tolerance`).
  - `delay`: The time the program should wait before continuing the execution when the condition is met.

If the condition is not met, the program will wait forever until the condition is met and then wait for the delay time before moving to the next step. If the device does not respond properly, the program will alert the user and fall back to the simple wait for time condition. However, the program will continue monitoring and trying to communicate with the device, so if any value is returned, it will revert to the original condition. This behavior could be changed in the future to be more rigid or more flexible, depending on the user's needs.

5.2.1.3.3 Scan

- `step`: Since it's a built-in instruction, the name of the instruction is `Scan` to invoke this special function.
- `metrics`: The list of metrics that should be performed sequentially for each iteration.
  - `step`: The name of the instruction that should be executed, it can be a simple instruction or a wait for instruction. It is important to always define a wait for step at the end of the metrics scan. Otherwise, the program will not know when the system reached the desired value, thus not waiting for it to iterate to the next step.

- `type`: The type of scan that should be performed. According to the stakeholder, the available options should be: `settle` and `sweep`. Both are linear functions but the `settle` type should wait for the metric to reach the desired value before running the measures once. On the other hand, the `sweep` type should keep the measures running while the metric is changing.
- `parameters`: The parameters of the scan.
  - `variable`: The name of the variable that should be changed.
  - `start`: The start value of the variable.
  - `stop`: The stop value of the variable.
  - `step`: The step value of the variable. If `start` is greater than `stop`, the step should be negative.
- `datafile`: The file path of the datafile that should be used to store the results of the scan. If not defined, the measures results will not be stored. The filepath can be a template string that will be rendered with the available variables, such as the pipeline name, the start time of the pipeline, and the variable name. More variables could be added in the future.
- `measures`: The list of measures that should be executed for each step of the scan.
  - `step`: The name of the instruction that should be executed. This could be a simple instruction or another scan instruction, making it possible to nest scans. In those cases, the dynamic datafile path can be useful to store the results of each step of an outer scan in a different file.

The scan instruction was designed to be as flexible as possible, allowing the user to define the scan type, the metrics that should be monitored, the measures that should be executed, and the datafile that should be used to store the results. In order to make it as versatile as shown, it is also quite complex verbose. The user should be aware of the possible configurations and the consequences of each one.

5.2.2 Deserialization

As explained in 4.2.1, the deserialization process is responsible for reading the input files and converting them into data structures that can be used by the software. Therefore, two steps are required: defining the data structures and creating the deserialization functions.

The data structures were defined using structs and enums to represent the objects in the input files, as it was explained in 5.2.1. Structs are used to represent objects that have multiple fields or key-value pairs (not the same as dictionaries) and enums are used

to represent objects that have multiple types or states. Besides that, both structs and enums can be defined with generics in Rust. Generics, as the name suggests, allow to use of the same object with different types. This is a useful and powerful feature for scalability and flexibility, but it can get way more complicated to code it right. Because of the Rust borrow checker, the developer is forced to think about the lifetimes of the objects and the references that are being used. This is a good practice and that is why Rust is considered a safe language, but it also makes the language harder to learn and use for beginners.

The following code snippet shows the `Device` struct that was defined to represent the device object in the input files. The snippet was simplified to show only the parts that would be relevant to the explanation.

```
#[derive(Debug, Clone, PartialEq)]
pub struct Device<Protocol>
where
    Protocol: Query
{
    pub name: String,
    pub description: String,
    pub instructions: BTreeMap<String, Instruction>,
    pub protocol: Protocol,
    pub default_arguments: BTreeMap<String, Arguments>,
}
```

At the top, we can see what is called derive macros and they can easily generate create some functions for the specific struct/enum. Next, the `Device` struct is defined with a generic, `Protocol`, and it uses the protocol attribute. The `where` section is used to define the constraints of the generic, in this case, `Query` trait. The `Query` trait is a custom trait that will be later explained in the communication section 5.2.3.

Using the Serde framework, the usual process to create the deserialization functions is using a derive macro in the object definition, automatically generating the functions for the developer. Although the derive macro offers some parameters to customize the deserialization process, it is not always enough. In this case, the input files are complex and the deserialization process is not straightforward and the macro is usually not capable of handling generic types, for instance.

Because of that, the deserialization functions were created manually following the Serde documentation. Moreover, this is a questionable architectural choice, some business logic was also added on these functions. The fact that the input files are also checked for inconsistencies and malformating is great and definitely a must but the deserialization process is probably not following the single responsibility principle. However, this was

considered a good approach for the present moment, as it would guarantee that no invalid input files would be used in the software.

In terms of what are these checks, the input files are checked for the following but not limited to:

- Missing mandatory fields;
- Commands and responses not well formatted based on the required parameters;
- Devices incapable of executing the asked instruction;

5.2.3 Communication

The communication process is responsible for sending commands to the devices and receiving responses their responses. To achieve that, a trait named `Query` was defined to represent the communication interface. It has a single function, `query`, that is used to send a command to the device and it can return a response if the used `command` expects one. The `query` function is asynchronous, meaning that it will not block the main thread while waiting for the response.

```rust
/// A protocol must implement the Query trait which allows
/// the device to send commands and receive responses.
#[async_trait]
pub trait Query {
    /// Send a command to the device and return the response (if any).
    async fn query(
        &self,
        command: &str
    ) -> Result<Option<String>, std::io::Error>;
}
```

The `async_trait` macro is used to define the trait as asynchronous, during the early stages of development, standard Rust was not capable of defining asynchronous traits. Nowadays, it seems to be possible to achieve the same result without this library but it requires more setup and boilerplate code. With that, the `Query` trait is defined with one asynchronous function, `query`, that receives a command string slice and returns a result with an optional string.

With that, a developer can easily implement the `Query` trait for any communication protocol that works with these requirements. For this project thus far, only the TCP protocol was implemented. In Rust, it would start like this:

```
#[derive(Debug, Clone, Deserialize, PartialEq)]
pub struct TCP {
    /// IP address of the device
    pub ip: IpAddr,
    /// Port of the device
    pub port: u16,
}

#[async_trait]
impl Query for TCP {
    #[tracing::instrument]
    async fn query(
        &self,
        command: &str
    ) -> Result<Option<String>, std::io::Error> {
        // Send the command to the device and return the response
    }
}
```

First, the `TCP` struct is defined with the `ip` and `port` fields. The `Query` trait is implemented for the `TCP` struct, defining the `query` function. The `query` function is annotated with the `tracing::instrument` macro, which is used to log information about the function execution using the Tracing library.

For the TCP protocol, the `query` function opens a stream connection, using the Tokio net submodule, with the device and then sends the command to it. After that, it waits for a response and stores it in a buffer. The response is then parsed and returned as a string, if the buffer is not empty.

5.2.4 Instruction

The instruction step, as shown in 5.2.1.3, is responsible for executing single instructions defined in the input files. In the program, there is a main loop for each step in the pipeline that will execute the instruction. If the program detects that the step is of type instruction - through an enum - it will execute the following logic:

1. Get the device object based on the device name;
2. Get the instruction object based on the instruction name;
3. Merge the available parameters with the parameters stack;

4. Render the query;
5. Send the query to the device;
6. Parse the response;
7. Return the response.

Steps 1 and 2 are used to search and get the equivalent object definitions for the device and instruction based on their names. During the deserialization process, the program verifies if these names exist, so this step should not fail at this point.

Step 3 introduces a new concept that will be used in all pipeline steps. The parameters stack is a data structure that will be used to store the parameters that are available for the current step. Since the pipeline can be defined with some special functions that can be used to set in-scope variables and also have functions inside functions, the parameters stack is a great strategy to keep track of the available parameters for each instruction call. So, in step 3, the previous parameters stack is merged with the parameters defined in the instruction step.

Step 4 is used to render the query string that will be sent to the device. The Tera library is used to format the query string, replacing the parameters in the query string with the values in the parameters stack. For more advance purposes, Tera can also be used to define control structures in the query string, such as if statements and loops.

Step 5 is used to send the query to the device using the `Query` trait. The `query` function is called with the formatted query string and the response is stored in a variable.

Step 6 is used to parse the response from the device. The response is a string that should be parsed to extract the values from it. The Regex capture groups feature is used to match patterns in the response string and extract the values from it. By defining the expected format as well as the data types for each variable available in the response, the program can "decipher" even some complex responses. For instance, it would be able to properly parse a response like `Temperature: 25.5°C` and return the value `25.5` as a float and the unit `°C` as a string.

Step 7 is used to return the response to the main loop. Depending on the stage of the pipeline, the response could be used to set a variable, to check a condition, or to store a value in a datafile.

5.2.5 Wait For

The wait for step, as shown in 5.2.1.3, is a built-in special function that is responsible for waiting for a metric to reach a certain value or time. The program will create a while

loop that will keep executing a certain metric or instruction, if set with one, in a certain interval until one of the conditions is met. The conditions are:

1. The wait for step is fully defined which means that a metric instruction is set. The function will keep executing the metric instruction in a certain interval until the metric reaches the desired value plus or minus the tolerance, starting the delay timer. If the metric is still between the tolerance range after the delay, the loop will break. If at any point the metric is not in the tolerance range, the delay timer will reset and wait for the metric to reach the desired value again.

2. The wait for step is defined but no metric instruction was set. In this case, the delay timer will start and if the delay is reached, the loop will break. This is similar to what programming languages usually call a sleep function.

3. The wait for step is defined but the metric instruction was set and the device did not respond properly. In this case, the delay timer will start and if the communication is reestablished, condition 1 will be checked. If the delay is reached, condition 2 will be satisfied.

The rest of the logic is similar to the instruction step, as explained in the previous section, 5.2.4. The main reason for the wait for step is to allow the user to set a condition required for the next step of the pipeline or, as we will see in the next section, the iteration of the scan step. For instance, as explained in 3.1.1, the cryomagnet can only operate in high magnetic fields if the temperature is equal to or below the superconductor critical temperature. That way, as a safe measure, the user could set a wait for step to check if the temperature is below the critical temperature before setting the magnetic field to the desired value.

5.2.6 Scan

Last but not least, the scan step, as shown in 5.2.1.3, is also a built-in special function that is designed to perform a scan through a range of values of on a list of metrics and measure different values for each iteration. This step is more complex than the previous ones and it is also really important for the users, as it is the heart of any scientific experiment: collecting data points for a system under different conditions.

A scan has 3 main components: the variable, the metrics, and the measures. The variable is the parameter that will be changed in each iteration of the scan, ranging from a start value to a stop value with a certain step. Moreover, this variable value will be used to update the parameters stack, so the metrics and measures can use it. The metrics are the instructions that will be executed in each iteration of the scan, basically setting the system

to the desired condition. The measures are the instructions that will be executed in each iteration of the scan, collecting the experimental data for posterior analysis. It is important to mention that the metrics and measure instructions are nested in the variable loop, so they will repeat for each iteration of the variable. One important difference between the metrics and measures is that the metrics are executed sequentially, you can think of them as an inner pipeline, while the measures are executed asynchronously, so we can query multiple instructions simultaneously.

The general steps for the scan can be summarized as follows:

1. Create the datafile if it was defined;
2. Set the variable to the start value;
3. Enter the loop;
4. Update the parameters stack with the variable value;
5. Execute the metrics;
6. Execute the measures;
7. Write the measures' results to the datafile;
8. Go to the next iteration (step 4).

The scan step is divided into two branches: settle and sweep. The settle branch is used to wait for the metric to reach the desired value before running the measures once. The sweep branch is used to keep the measures running while the metric is changing. If we look at the general steps, the settle branch will execute steps 5 and 6 sequentially, while the sweep branch will execute steps 5 and 6 in parallel. This means that the measures will run in a loop until the metric reaches the desired value, then repeat the cycle for the next iteration of the variable.

During development, Milton Tumelero, the advisor of this work, provided a couple of useful experiments that the lab would run in the cryostat. One of these examples called "magnetoresistance" requires two inner loops: one to set the temperature and another to set the magnetic field. Because of that, the scan step was designed to be flexible enough to allow nested scans. In order to achieve this, the user can define a scan step inside the measures array of another scan. This way, the program will execute the inner scan for each iteration of the outer scan, allowing the user to perform complex experiments with multiple variables. In light of this, the datafiles names were also upgraded to allow template strings that will be rendered with the available variables, so that a new file can be generated for each iteration of the scan.

6 Results and Conclusions

6.1 Results

The developed software was tested against all the available devices mentioned in 3.2 being able to communicate with them and execute simple instructions. The easiest way to verify if the communication was working was to send the identification command (*IDN?) to the devices and check if a response was received in a reasonable time. The following lines will demonstrate a sample pipeline to run this test:

```yaml
name: Check communication
description: Run a simple test to check if the communication with the devices is working.

devices:
  - path: ./config/devices/TM612.yaml
  - path: ./config/devices/24C.yaml
  - path: ./config/devices/4G.yaml

pipeline:
  - step: Device instruction
    device: TM612
  - step: Device instruction
    device: 24C
  - step: Device instruction
    device: 4G
```

It was noticed during these simple tests that the LabView program was not releasing the devices properly, even when it was paused/aborted. This is probably due to the way the LabView program was implemented or the way it handles communication with the devices. As explained in 5.2.3, the developed solution creates a stream for each query and closes it after the response is received. This way, although it may add some overhead to the communication, it ensures that the devices are not blocked by the program, especially if it is improperly closed.

Other simple commands were tested, such as retrieving the temperature from the temperature monitor and setting the temperature setpoint in the temperature controller.

Currently, as a test, when an instruction step is used, it will print the response from the device to the console.

The wait for step was also tested and it worked as intended for all 3 possible conditions, as explained in 5.2.5. Since the verification is performed at a high frequency, the response of each command is not prompted to the user in the console, although it is stored in the log files. However, when the delay timer starts or is reset, a message is printed to the console. During the tests, and this is a known issue with the current LabView program as well, the communication with the devices could miss some responses. However, the program is capable of handling this situation using the third condition of the wait for step. A sample pipeline to test the wait for step is shown below:

```yaml
name: Wait for the temperature to stabilize
description: Wait for the temperature to stabilize in the temperature
↪ controller at 4K.

devices:
  - path: ./config/devices/24C.yaml

pipeline:
  - step: Wait for
    metric:
      instruction: Get temperature
      device: 24C
    parameters:
      name: temperature
      value: 4
      tolerance: 0.1 # 3.9 K < temperature < 4.1 K
      delay: 120 # 2 minutes
```

The scan operation was also tested and it was able to loop through the specified range of values and run the metrics for each step, preparing the system to a particular state. As it was explained in 5.2.1.3, this usually required at least two steps in the metric block, one to set the value and another to wait for the system to stabilize.

Unfortunately, the measures block was not working properly at the time of this work submission. The measures block was supposed to run a set of gather instructions, after the metric block in case of of the “settle” type or during the metric block on every interval in case of the “sweep” type, and store the results in a file. This section of the program was not working properly due to some issues with the asynchronous functions in Rust, which is known to be one of the hardest parts of any programming language to

work with. Therefore, due to time constraints, it was not possible to fix this issue in time. A sample pipeline to run a scan operation is shown below:

```yaml
name: Scan temperature
description: Ramp the temperature from 10k to 300K in steps of 10K.

devices:
  - path: ./config/devices/24C.yaml

pipeline:
  - step: Scan
    metrics:
      - step: Set temperature
        device: 24C - Sample
        # temperature value will be defined by the scan parameters
      - step: Wait for
        metric:
          instruction: Get temperature
          device: 24C - Sample
        parameters:
          name: temperature
          # value will be defined by the scan parameters
          tolerance: 0.1
          delay: 120
    type: settle
    parameters:
      start: 10
      end: 300
      step: 10
    datafile: ./data/temperature_scan.csv
    # Currently, the measures block is not working
    # measures:
    #   - instruction: Get temperature
    #     device: 24C - Sample
```

6.2 Requirements Evaluation

Based on the results obtained until the time of this work submission, explained in the previous section 6.1, we can evaluate which goals, 1.2, and requirements, 2.3, were met.

It is fair to say that mostly basic goals, 1.2, were fully met, except the third item which mentions “storing the data for posterior analysis”. As it was shown in the previous section, the measure block that is part of the scan operation is not working as intended. Aside from that, the program is capable of communicating with all the proposed devices attached to the cryostat; it is capable of sending any type of instruction to them, controlling its state; and the program is modular and flexible enough to be extended to work with different equipment and protocols.

Finally, there were 6 requirements, 2.3, that were identified by the laboratory representative, Professor Milton Tumelero. The first two, *Read operations* and *Set operations*, were fully met through the instruction step, 5.2.4. The *Wait for* requirement was also achieved as intended, as it was detailed in 5.2.5. The *Scan* requirement was partially met, as the program is capable of looping through a range of values, running the metrics for each step, but the measures are not working properly. The *Run on a Windows machine* requirement was also met, since a Rust program can be compiled to run on Windows natively or through cross-compilation. Finally, the *User-defined pipelines* requirement was fully met, as the program is capable of running a sequence of operations defined by the user as shown in 5.2.1.3.

In conclusion, the proposed solution is feasible and it is capable of performing the main operations it was designed for. Some tweaks and improvements are still needed to make it useful for the laboratory but it can be considered a successful Minimum Viable Product (MVP). In order to make it ready for production for early adopters, the measures piece of the program needs to be fixed and some other improvements could be made, as suggested in the next section.

6.3 Future Work

6.3.1 Measures Block

In case the project is continued, the first and most important task is to make the measures block work properly. This is the main feature that is missing to make the program usable in the laboratory and check all basic requirements. The problem may not be complicated in terms of how many lines of code need to be changed, but it requires a better understanding of how async works and how to properly set up runtimes or threads for it.

6.3.2 Scoped Variables and Attributes

Next, a known possible problem for more complex pipelines is the repetition of the same variable names which can lead to conflicts. Furthermore, this has already proven

to be a problem in the current implementation while working with the measures block. For example, if a scan operation has two measures that return the temperature, which is quite possible, if the program simply stores the results in a variable called `temperature`, one measure could overwrite the other result. Currently, the proposed solution was to also include the device name for that particular scenario but it is not a scalable solution. A better approach would be to scope the variables and attributes better, or even let the user define aliases for the variables.

6.3.3 Configuration File

Another suggestion for future work is to create a configuration file to grant the user more flexibility over general aspects of the program. For example, the user could define the interval between measures, what to do if a measure fails, or even the timeout value for communication with the devices. This would improve the experience for more advanced users and make the program more suitable for different workloads.

6.3.4 Meta Instructions

During the usage of the program, it was noticed that some pipelines could get quite verbose, especially considering that some operations are guaranteed to be repeated in different pipelines. For example, a scan operation that ramps the temperature will usually interact with one device to set that temperature and maybe the same one or another to get the temperature and wait for it to stabilize. To avoid repeating this same template in different pipelines, a suggestion is to create custom functions with a set of parameters to use in other pipelines. For example, a `scan_temperature` function specifically designed for the cryostat could accept a minimum set of parameters, such as the start, end, and step values, and measures to run during the scan. That way, instead of writing the scan step from scratch, the user could simply call the `scan_temperature` function with the desired parameters.

6.3.5 Graphical Interface

Finally, a graphical interface could be developed to make the program more user-friendly. This would be especially useful for users that are not familiar with the command line or the YAML syntax. Not only that but a graphical interface could lead to less errors in the pipelines, since the user would not need to remember the exact syntax of the program. This would also make the program more accessible to users that are not familiar with programming or the command line.

6.4 Highlights

To conclude this work, it is important to highlight the main points of the project and its contributions. The goal of this project was to develop a program capable of controlling and monitoring devices linked to a cryostat, while allowing the user to define custom pipelines to automate experiments. Not only the program was able to communicate with the devices and execute the main operations it was designed for, but it could also be extended to work with different equipment and protocols. Thus, proving that the proposed solution and software architecture are feasible to solve the initial problem and more.

Compared with the existing LabView program, described in 2.1 and 2.2, the developed solution has some advantages and disadvantages, as shown in the table 2.

Feature	LabView	Developed Solution
Flexibility	User must modify a significant portion of the code to perform different tests.	User-defined pipelines allow the user to run an arbitrary sequence of operations.
Scalarity	Limited to work with the cryostat. Moreover, in order to perform different tasks, entirely new programs are created.	Modular and flexible enough to be extended to work with different equipment and protocols.
Price	Proprietary and expensive software.	Open-source and free of charge.
User-friendly	LabView is more intuitive and easy to use since it is a graphical programming language. Moreover, the current program has charts to keep track of the collected data.	The developed solution is a command-line program, which may not be as user-friendly as LabView, but it can be extended to have a graphical interface.

Table 2 – Comparison between the existing LabView program and the developed solution.

Moreover, it was possible to verify by this work that Rust can be used to create complex instrumental systems solutions. As it was explained in 5.1.1.5, Rust is a modern and safe programming language that provides some trivial and non-trivial advantages over other languages, such as performance, lower memory footprint, and being less prone to errors. The latter is especially important for this project and it deserves to be emphasized. Rust's type system and ownership model make it very clear to the developer where the program could fail, letting the developer handle those cases properly. In other programming

languages, such as Python, it is not always clear what a function can return or raise, which can lead to unexpected behavior and runtime errors that can be unacceptable in some conditions this software may be used. In various ways, Rust teaches the developer to write better code, following best practices and avoiding common pitfalls.

Finally, this project not only provides a solution for the initial target problem but also serves as a proof of concept for future projects using Rust for similar applications. Despite Rust's steep learning curve, it has shown to be a great choice for this project and it is recommended for future projects with similar purposes.

References

1 CRYOMAGNETICS, INC. *Operating Instruction Manual for Superconducting Magnet System with Variable Temperature Insert*. English. Rev. 1. 30 Aug. 2015. 31 pp. Cit. on p. 17.

2 CRYOMAGNETICS, INC. *User's Guide Models 612 and 614 Cryogenic Temperature Monitor*. English. Rev. 1. Mar. 2014. 119 pp. Cit. on p. 17.

3 CRYOMAGNETICS, INC. *User's Guide Model 24C Cryogenic Temperature Controller*. English. 2014. 227 pp. Cit. on p. 18.

4 CRYOMAGNETICS, INC. *Operating Instruction Manual for 4G Magnet Power Supply*. English. Rev. 9.0. 17 Feb. 2010. 101 pp. Cit. on p. 20.

5 STACK OVERFLOW. *Stack Overflow Developer Survey 2024*. Available from: <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>. Cit. on p. 30.