Electronic Communications of the EASST
Volume 85 Year 2025

deRSE25 - Selected Contributions of the 5th Conference for Research Software Engineering in Germany

Edited by: René Caspart, Florian Goth, Oliver Karras, Jan Linxweiler, Florian Thiery, Joachim Wuttke

Experiences from Bottom-Up Python Software Development in Experimental Physics

Yudong Sun, Mingsong Wu

DOI: 10.14279/eceasst.v85.2628

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Experiences from Bottom-Up Python Software Development in Experimental Physics

Yudong Sun[1,†], **Mingsong Wu**[2,†]

[1] yudong.sun@physik.lmu.de
Fakultät für Physik, Arnold Sommerfeld Center for Theoretical Physics (ASC)
Munich Center for Quantum Science and Technology (MCQST)
Ludwig-Maximilians-Universität München, Germany

[2] mingsong.wu@unige.ch
Quantum Technologies Group, Geneva Quantum Centre
Université de Genève, Switzerland

[†]These authors contributed equally

Abstract: Automation has become an important aspect of effective and rigorous experimental research work, particularly in physics. Among the diverse automation platforms, Python stands out with its large repository of open-source scientific software for instrument control and data analysis. However, despite considerable effort from the research software community to unify multi-instrument control (also known as instrument orchestration) and to make it laboratory-agnostic, there has yet to be a strong consensus on a universal package that is easily adaptable. We contextualise this issue by discussing the underlying barriers that we have encountered within the experimental physics community, such as the unfavourable circumstances for software development, and the disparity in programming skills amongst physicists. A sustainable way forward could be specialised but well-maintained software repositories within each research group. In that spirit, we present our experiences as case studies on building software for experiments and share key coding considerations that may be helpful to other physicists.

Keywords: Python, Modularity, Scientific Instruments, Orchestration, Automation, Data Acquisition, Data Visualisation, Interfacing, Hardware-Software Interaction, Graphical User Interface, Code Sustainability, Experimental Physics

1 Introduction

Rigorous measurements form the cornerstone of experimental physics research. As experimental setups grow in complexity, so does the need to automate data collection and analysis to ensure the repeatability of the results. Automation in this form likely developed organically in many different laboratories simultaneously and lends itself particularly well to labour-intensive and/or repetitive tasks [Ols12].

Since such processes often involve the automation of equipment and instruments, this is also sometimes called **instrument orchestration** [RFG+22] or **experimental control systems** [GFJ+04]. In its most basic form, automation involves some way of extracting data from a sensor or measurement device in a way that allows the data to be collated without the manual labour of writing the data down. For example, if an instrument only has a digital display output (e.g. a digital multimeter), one could use a webcam and some image-recognition software to extract the readout to a computer for processing. Figure 1 (left) depicts an example of a simple but functional setup being used in field research conducted in the Arctic [NP24], juxtaposed with Figure 1 (right), which depicts an advanced experiment control system at CERN/ATLAS experiment used to collect and process the large amounts of data generated [CER25].

Despite the importance of such automation, the authors note that in the publication of physics research works, the associated software is usually relegated to supplementary material or not published at all, available only by explicit request to the authors. Furthermore, often only the code used for data *analysis* and not the code for data *collection* would be provided. This is perhaps similar to how one would usually not publish the alignment process of an optical setup used to obtain a measurement.

The need for better software in instrument orchestration is not new [Sun23]. It remains common to encounter convoluted, cryptic code that somehow is still running the same experiment after a decade. Since the code can be so difficult to understand, maintenance and continued development are put off as much as possible. A survey by [Het18] reports that 21% of people developing software in research have never received any form of training in software development. In that respect, we hope that this contribution can provide some clarity, in particular when writing custom software, thereby helping to improve the reproducibility [KCAC20] and the reliability [Het14] of software and its results, especially with respect to automation and instrument orchestration.

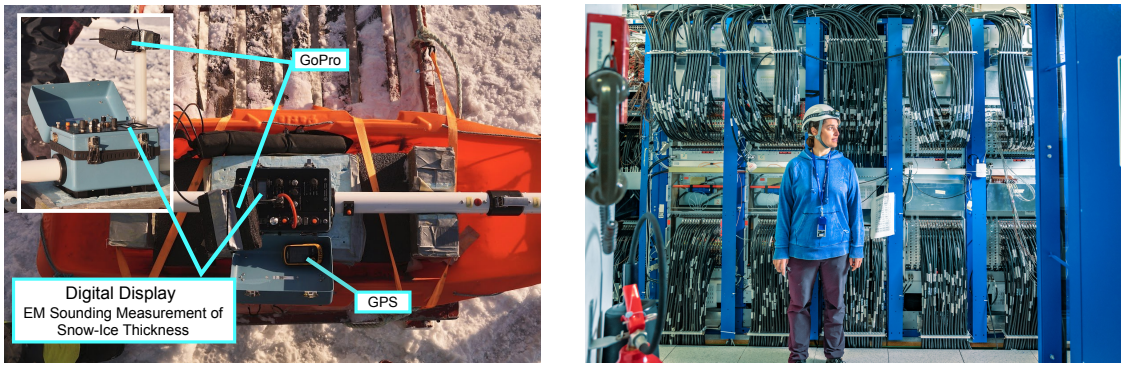

Figure 1: **(Left)** A rudimentary form of instrument orchestration: A GoPro camera is depicted here being used to record data from the digital display of a Geonics EM31-ICE for snow-ice thickness measurements. The footage is matched to GPS data to build a location-resolved data series [NP24]. Illustration adapted from [Fou24b]. **(Right)** In comparison, an advanced experimental control system at CERN/ATLAS, the Trigger and Data Acquisition system, which is used to pick out distinguishing events and reduce data flow [CER25]. Photo from [CG20].

Being among the few (if not only) members of the experimental photonics community at the deRSE24 conference, we were motivated to share, with this contribution, our observations and experiences writing and designing software in our community with the larger RSE and experimental physics community.

1.1 Problem Statement and Goal

There is a large diversity of solutions in experimental physics that address the complexity of instrument orchestration, with some research groups requiring highly niche implementations. While some turn to hardware-based solutions (analogue or digital electronic circuits, FPGAs, etc. such as [TZP21, Jä24]), software-based automation remains a popular choice among experimentalists. Using software lowers barriers of entry in terms of equipment and the necessary technical knowledge required to set up and operate the equipment. We make the following observations regarding building custom software as experimental physicists:

1. **Python is Ubiquitous in Physics**: Being one of the most popular programming languages in the scientific community [HBC+22], most people entering research already have some Python programming knowledge.

   Python also stands out as an open-source, cross-platform language with a vibrant developer community and a large repository of useful scientific packages [Jtv15]. Furthermore, the class-based structure of Python maps particularly well to the way that physicists deconstruct research problems, making it an intuitive programming language to work in [SH15].

2. **LabVIEW is Popular but has Strong Disadvantages**: LabVIEW by National Instruments has been a particularly long-standing and popular program within experimental physics for hardware automation. It utilises the concept of *data-flow* instead of *imperative* programming used in Python.

   While LabVIEW is designed for instrument orchestration [Kod20], it is closed-source and uses a graphical (non-textual) programming language, making version tracking/control difficult (also cf. Subsection 4.4). LabVIEW programs are also difficult to share and collaborate on: The lack of forward compatibility between different versions of LabVIEW [Han] combined with the expensive proprietary licensing [Nat24] can also increase the barriers to creating a sustainable LabVIEW ecosystem. Complex data analyses that can be quickly implemented in Python would require a highly complicated LabVIEW program.

   Hence, while LabVIEW can be useful in some situations (especially when it comes to rapid hardware deployment and visualisation), it could result in substantial overhead in software management that disincentivises sustainable software.

3. **A Single Standard is Difficult**: As shown by the continual appearance of independent drop-in Python packages over recent years [RFG+22, KCAC20, Jtv15, Web21], there has not been a consolidated standard and community on instrument orchestration in experimental physics. This can be broadly attributed to the diversity of research goals, experiment layouts, and equipment availability.

With each lab's infrastructure being unique, custom software tends to be built organically, in other words bottom-up, with automation code for unique individual experimental segments being developed before any high-level orchestration planning is done. We believe that aiming for a one-size-fits-all standard is not realistic within the research context.

4. **Programming Capabilities Vary**: There is an intrinsic selection bias in published code, where researchers who publish software suites tend to be of the minority that already possess specialist resources and knowledge to package and maintain a healthy code ecosystem (e.g. a Research Software Engineer, or RSE for short). It is thus not obvious that these successful examples can be easily adapted and maintained for use by other researchers.

   The difficulty in retaining knowledge of experimental control code infrastructure is also well-known given the high turn-over rate of academic staff [Web21]. Without a proper handover, a group may lose expertise in both the usage and the design of custom Python packages. While some groups find it sufficient to pass down only the high-level usage of Python packages, this could lead to issues with debugging and adapting the code in the future. This can quickly become unsustainable if basic Python literacy is not ensured across generations of researchers.

5. **Sustainable Research Software in Physics is Not Incentivised**: As mentioned, the lack of recognition of the work done on custom research software for generating and processing important scientific results is an ongoing issue in academia [Het16]. This is further compounded by inadequate software training amongst researchers despite the need for research software [Het18]. Anecdotally, explaining software used is rarely required (but potentially allowed) in physics journal submissions or their supplementary materials, as it is the analysis of the experimental results that reveals novelty in the work which then determines acceptance. We believe this situation to be a major factor for poor coding practices and thus unsustainable software in experimental physics.

With this in mind, this contribution aims to provide recommendations for building a sustainable code ecosystem for an experimental physics laboratory while recognising the practical circumstances of coding as a physicist. We wish to address physicists with basic Object-Oriented Programming (OOP) understanding who do not necessarily have a software engineering background or access to the services of an RSE. While we do mention Python packages, including our own, they are only examples of how to incorporate desirable features: Our focus is **not** to prescribe the adoption of any particular package and instead to encourage our audience to take certain design inspirations while developing their code, or at the very least, make writing sustainable code seem less daunting than it might seem at first.

In Section 2, we look at the tools and references that can guide us when we build our own software. Section 3 then presents the authors' experiences developing software in experimental photonics and quantum optics to provide some recommendations for physicists trying to build and maintain software ecosystems for their groups. Finally, Section 4 lists some key design considerations that we have distilled from our experiences and found most helpful, followed by the conclusions in Section 5.

2 Resources and Good Practices for Building Software in Experiments

In this section, we present the available resources we utilise for both implementing and taking inspiration for our software. These concepts and tools will also be mentioned repeatedly in the following sections.

2.1 Device Interfacing Standards

Data acquisition from scientific instrumentation can often be characterised by rudimentary methodologies, as illustrated in Figure 1. Indeed, the utilisation of decades-old equipment incorporating obsolete technologies remains prevalent in contemporary experimental configurations. This is partly explained by budgeting considerations within each research group, where not all equipment are given equal priority for upgrades.

Fortunately, the standardisation of hardware interfacing protocols has mitigated some of these challenges. Two predominant standards have emerged: the Standard Commands for Programmable Instruments (SCPI), introduced in the 1990s [SCP99], and the Virtual Instrument Software Architecture (VISA), proposed in 1995 [VXI24]. SCPI constitutes a set of standardised commands instruments can recognise, while VISA facilitates device intercommunication irrespective of the underlying physical connections, such as USB or ethernet.

The establishment of these standards, in conjunction with the development of Application Programming Interfaces (APIs), has significantly enhanced the efficiency of software development for instrument control. When designing custom software for laboratory equipment communication, we often first seek to leverage these device-interfacing standards to aid the development process and future-proof software as much as possible.

2.2 Good Practices: FAIR Principles

The FAIR Data Principles constitute a framework originally designed for enhancing the long-term reusability of scholarly data. These guidelines were created by a diverse working group representing both private and public sectors, with particular emphasis placed on automating data retrieval via standardised data management practices [Wil16]. The idea of "FAIRness" has since been adapted for research software by the FAIR for Research Software (FAIR4RS) working group, with the goal being to similarly enhance the longevity, access, and reusability of software used to produce scientific results [BCK+22]. We strongly resonate with FAIR4RS's mission and indeed find FAIRness a useful benchmark for evaluating the sustainability of our software:

- **Findability**: This refers to the ease with which software can be located. Key practices include assigning unique identifiers to every package/component version and publishing software in recognised repositories (e.g., GitHub, GitLab).
- **Accessibility**: The ability to retrieve the software from a standard communication protocol. This involves providing authentication and authorisation protocols where necessary, comprehensive installation instructions, and persistent metadata even when the software is no longer available.

- **Interoperability**: This pertains to the software's ability to exchange data and integrate with other systems. Key considerations include using well-documented file formats, modularity in software design, and providing clear user programming interfaces.
- **Reusability**: Reusability in software extends beyond accessibility, emphasising the ability to adapt the software for different contexts. Practices promoting reusability include thorough documentation of code functionality and underlying algorithms and providing a clear usage license.

2.3 Open-Source Package Example: Bluesky

The Bluesky package [KCAC20] is an example of a state-of-the-art and open-source instrumentation package that addresses many of the challenges inherent in scientific data acquisition and instrument control. Originally developed for the synchrotron science community at the National Synchrotron Light Source II[1], this open-source Python package demonstrates sophisticated capabilities and has been adopted by several other facilities [ACCR19a]. We note the following attractive features:

- **Abstraction and Modularity**: The package implements several abstraction layers, both in instrument orchestration and hardware interfacing. In particular, the latter is achieved through a sister package called ophyd, as part of the larger experimental control ecosystem at NSLS-II. By having ophyd handle all hardware-level specifics, Bluesky can be agnostic to the type of equipment being deployed in an experiment. This allows similar experimental control scripts to be used across different setup configurations.
- **Extensibility**: Bluesky is built to be compatible with the hardware interfacing solutions provided by various supporting packages, such as ophyd and instrbuilder. These packages in turn leverage device standards within the high-energy physics community, namely EPICS and SPEC [ACCR19b] respectively. This lowers the barrier of entry for researchers to contribute more equipment drivers to the ecosystem.
- **Hardware Emulation**: The package offers hardware emulation capabilities for testing code without the physical hardware being available to the programmer. Given a large-scale experiment comprising thousands of devices (such as in particle accelerators), this can be a particularly useful feature.

On the other hand, in the context of experimental physicists trying to develop bottom-up solutions, Bluesky's feature-rich design may also present certain challenges:

- **Complexity**: The package's extensive feature set and advanced architecture may present a steep learning curve for new users, potentially limiting its *full* accessibility to researchers without strong programming backgrounds or the assistance of an RSE.
- **Specialisation**: As with many similar open-source instrument orchestration packages, Bluesky was originally designed for a specific community within physics. Hence, the

[1] NSLS-II, at Brookhaven National Laboratory, USA

package may have inherent design choices that are less optimal for other scientific domains. For example, the aforementioned device standards specifically cover equipment commonly found in synchrotron facilities.

- **Potential for Over-Engineering**: The impressive scope of Bluesky may in fact exceed the needs of many individual researchers or smaller labs, potentially introducing unnecessary complexity for simpler experimental setups. The layers of abstraction, while offering versatility in software design, may also cause significant computational overhead for laboratories without suitably performant computers.

Needless to say, Bluesky is a mature and extremely capable package for instrument orchestration, and also follows FAIR principles closely. Similar case studies include NICOS, which is also seeing rising cross-facility usage [MLZ21]. We see these packages as model examples showing how a comprehensive (albeit resource-intensive) instrument orchestration software suite can look like, even if research groups cannot import them directly.

3 Case Studies in Experimental Physics

In this section, we explore two projects that the authors have developed in experimental physics. These projects emerged organically in response to specific research needs within groups of modest sizes, and their development followed an iterative, bottom-up approach. In the next section, we aim to examine the concepts and considerations that were the most useful in helping our projects to be more sustainable and achieve their goals.

3.1 Nanosquared: Automated M^2 Laser Beam Quality Measurement Package

Characterisation is a crucial part of experimental physics. Proper characterisation ensures that experiments are accurate, results are reproducible and the underlying phenomena could be understood as much as possible.

In laser development, the M-squared $\left(M^2\right)$ beam quality factor is commonly used to characterise the quality of a laser beam. This factor provides a simple empirical way to assess the laser's effectiveness in various applications [Pas]. Since so many imperfect properties of the laser beam are summarised into this one factor, it provides a quick way to measure the quality of the laser across different laser technologies and use cases, albeit not without its downsides. The typical way of measuring the M^2 of a laser source involves [ISO21]:

1. Taking at least 10 beam-width measurements around the beam waist of a focused laser beam, and
2. Fitting the data points (beam-width against measurement position) to the Gaussian beam propagation equation with M^2 as one of the fit variables.

This was the context behind nanosquared[2], a package developed by one of the authors (Y. Sun) over the course of a year at the experimental laser physics group *Attoworld* at the Max-Planck-Institute of Quantum Optics (MPQ) in Garching, Germany. A series of custom short-pulse lasers

[2] https://github.com/sunjerry019/nanosquared, $\sim$ 9200 lines of code, $\sim$ 5 users within MPQ.

were being developed for use in the group's longer-term research campaigns, and a few key challenges arose:

- The development of numerous lasers and amplifier setups necessitated frequent performance characterisations; however, these repetitive measurements, as described above, quickly became *tedious* and *labour-intensive* when done manually, taking precious time away from physics research.
- No affordable commercial solutions were readily available for the lasers' uncommon wavelength (mid-infrared).
- The specialised beam profiler that was used in the group for beam-width measurement had a closed-source interfacing driver and only supported automation using a long-deprecated software framework called ActiveX. This was a particularly challenging obstacle even for physicists with programming experience, given the obscurity of ActiveX amongst modern instruments in experimental physics.
- The fitting process for M^2 is very sensitive to the input data and the initial guesses of the fit parameters due to the complex parameter space. Automating such a process would standardise the measurement process, thus increasing the *reproducibility* of the measurement

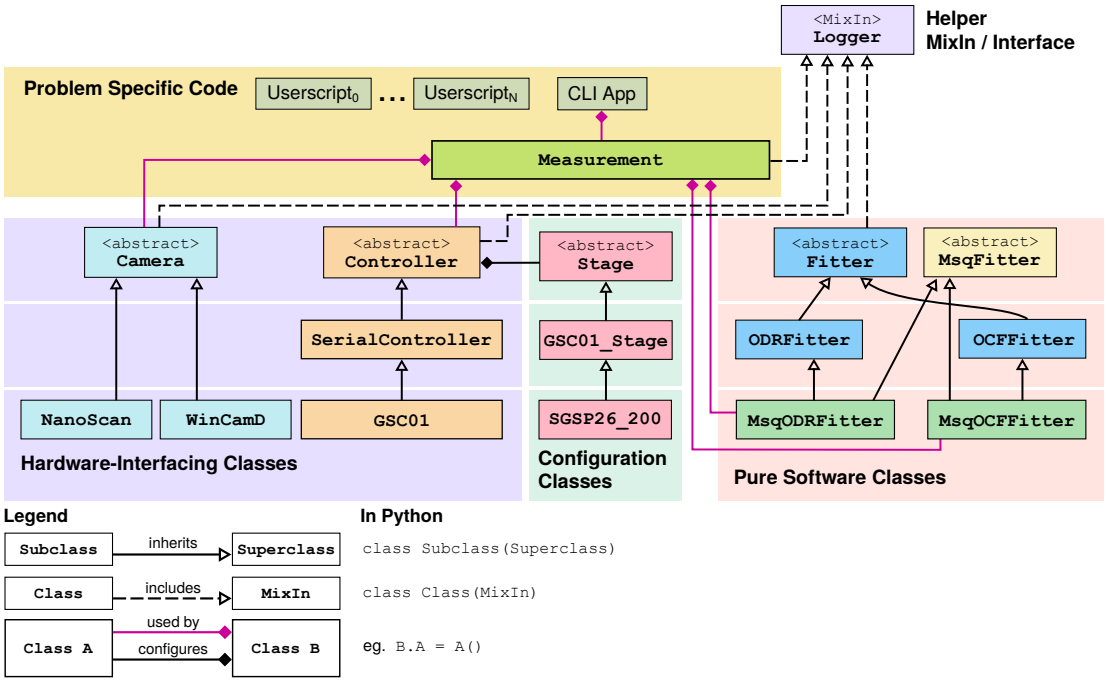

Figure 2: Modules of the nanosquared package. Classes could be conceptually categorised into hardware-interfacing, pure software, configuration, and helper. The mix-in class `Logger` has been put at the top so that inheritance always goes in the same direction graphically. $\text{Userscript}_{0..N}$ may be created that makes use of any of the individual modules below.

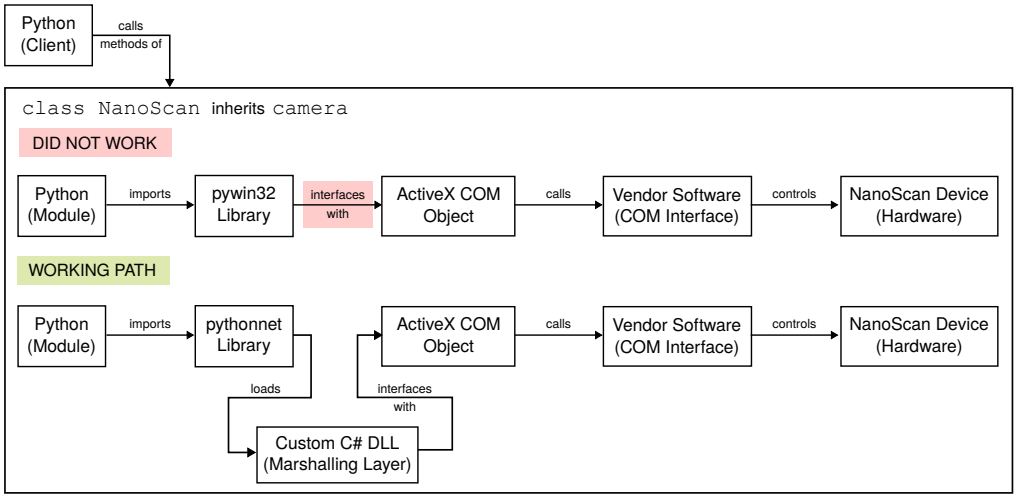

Figure 3: Software component interaction diagram of a specific hardware-interfacing workaround in the nanosquared package, with the call stack represented from left to right. The ActiveX COM module was incompatible with Python (this is represented by the red shaded box in the top path). As a result, an extended interfacing path through a C# DLL was necessary for Python to communicate with the NanoScan beam profiler device.

process. Furthermore, automation would also improve precision by increasing the number of samples that can be taken for the measurement, thereby reducing its uncertainty.

Naturally, software for the fitting workflow already exists in the group for manual M^2 measurements, so the task was to expand upon this to automate the measurement and fitting process. Existing code also acted as a baseline, allowing us to *verify* that the new workflow generated results that were compatible with earlier methods. The package was largely conceptually separated into the following components, with details depicted in Figure 2:

- Hardware interfacing (translation stages, beam profiler, etc.)
- Configuration classes (storing device model-specific settings)
- Pure software components (data processing and fitting)

This separation was done with a focus on modularity and separation of concerns. Each component could be and was independently developed and tested in experimental runs. The division of the code into these distinct components was also intended to establish clear, measurable milestones, facilitating progress tracking and sustaining project momentum.

Considerable focus was also placed on ensuring abstraction so that we describe *what* the code should do and not *how* it should do it. When properly implemented, this abstraction neatly hides much of the complexity inherent to interfacing software and hardware.

The hardware-interfacing classes are where this abstraction had the biggest benefit. As more precise measurements were required, the DataRay beam profiler [Inc24] that was initially used had to be changed to a NanoScan scanning slit beam profiler made by Ophir Optics [Ltd24].

Thanks to the abstracted "beam profiler" class that both devices share, the switch between device/manufacturers was simple, requiring no high-level changes to the larger experimental control program.

While developing the code for this change, we realised that the automation of the NanoScan beam profiler was only possible through a type of ActiveX COM Object. Unfortunately for Python, this type of ActiveX COM Object was not supported by popular libraries such as pywin32[3]. To resolve this, we opted to use C# (which supports this object type) as an interpreter between Python and the ActiveX COM Object that ultimately talked to the beam profiler. This is depicted in Figure 3. With abstraction, we were able to enclose this inelegant technicality concisely within low-level modules.

Alongside hardware-interfacing classes, configuration classes were created to manage multiple devices of the same type with the similar communication methods but different hardware specifications. These configuration classes enable flexible handling of device-specific parameters. For instance, devices like linear translation stages that use the same controller but have different ranges could be easily swapped within the system. This is accomplished by instantiating the appropriate device class (e.g., `StageB()` in place of `StageA()`) without altering the underlying communication logic.

The organisation of the pure-software fitting components of the project can be understood as the separation of fitting algorithms and models. This separation ensures that the various fitting strategies (classes) can be reused in the future to fit different models without requiring extensive modifications to the underlying code, which often remains the same anyway. Similarly, the component representing the model being fitted may be used with alternative fitting methodologies if necessary. This modular approach emerged due to disagreements over the choice of fitting method, as it was observed that the choice significantly influenced the final results.

Finally, a mix-in helper class `Logger` provides all classes with an easy and unified way of logging through the `self.log(...)` method. Leveraging Python's support for multiple inheritances, this helps to reduce code maintenance overhead and simplifies future extensibility to change logging behaviours, such as logging to a file instead.

These modules for the different components of an M^2 measurement setup were then brought together into a full Python package for the research group. A pleasant side effect of this project was that several existing and legacy stand-alone modules for specific instruments were integrated or refined in the process, which encouraged code reuse and facilitated code maintenance.

The Python package was then given a front-end in the form of a command-line interface (CLI) so that our colleagues could easily do a quick M^2 measurement without writing new code. Testing the software within our group, we concluded that the CLI was more than sufficient for our needs. We thus opted, contrary to our original plan, not to develop a graphical user interface (GUI) as it would have been unnecessary.

[3] The ActiveX COM object provided by the vendor had functions where parameters were passed-by-reference and not passed-by-values. This form of function call is not supported by Python, thus preventing direct interfacing.

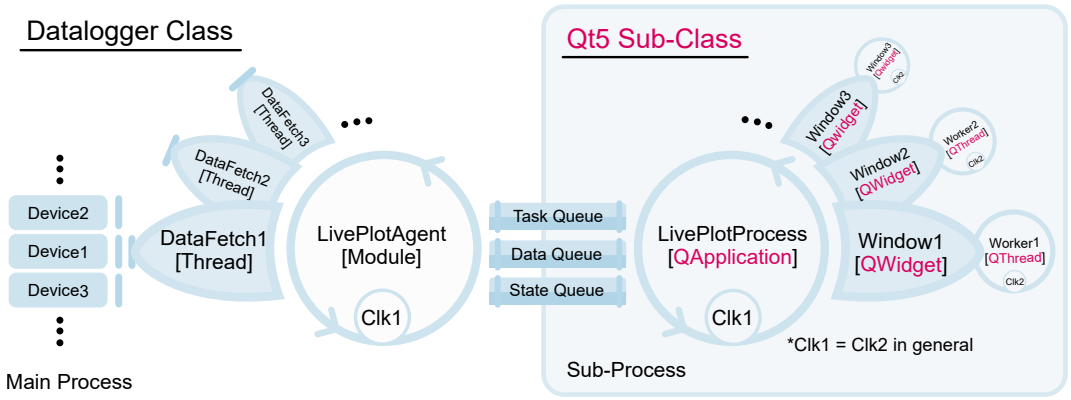

Figure 4: Concept illustration of liveplotter's operation. LivePlotProcess interfaces with all graphical objects and is contained within a sub-process, while LivePlotAgent exchanges instructions and data with other modules under the main Datalogger class using thread-safe queues.

3.2 Liveplotter: Generalised Live Data Visualisation Module

Live visualisation of data in experimental physics is a crucial tool that allows researchers to quickly identify desired signals or unexpected behaviours without post-processing the data. This immediate feedback enables on-the-fly adjustments to setup parameters and monitoring, thus improving work efficiency.

While some instruments already possess data displays by default, such as oscilloscopes and spectrometers, they tend to be expensive and specialised. Physicists often need to monitor live raw data from a variety of devices as well as combined data from multiple devices. This was the context behind liveplotter[4], a generalised live plotting module developed by one of the authors (M. Wu) [Wu24] over the course of a year to facilitate experiments in the Quantum Measurements Laboratory (*QMLab*) at Imperial College London. To perform quantum optics experiments, QMLab initialises several instruments at once with custom drivers all written in Python, which allows multi-instrument orchestration to be done from a single command line interface (CLI) within a Python virtual environment. This mode of operation offers a few key benefits, such as rapid code prototyping and versatility to equipment changes, but imposes some requirements on a live plotting solution:

- **Device-Agnostic**: With live data coming from many sources that are also constantly changing, a useful live plotting module must be generalised for several data formats.
- **Asynchronous**: As QMLab uses Python for live experimental control as well, the live plotting module must collect and display data as a background (non-blocking) process.
- **Drop-In**: Since experimental setups rapidly grow in complexity and change configurations, it was deemed impractical to develop and constantly adapt a GUI for experimental control with live plotting GUI elements. The live plotting module thus needs to behave like a drop-in module that can be used with a CLI.

[4] https://github.com/metrosierra/liveplotter, $\sim$ 600 lines of code, $>$ 10 users from QMLab and presently QTech.

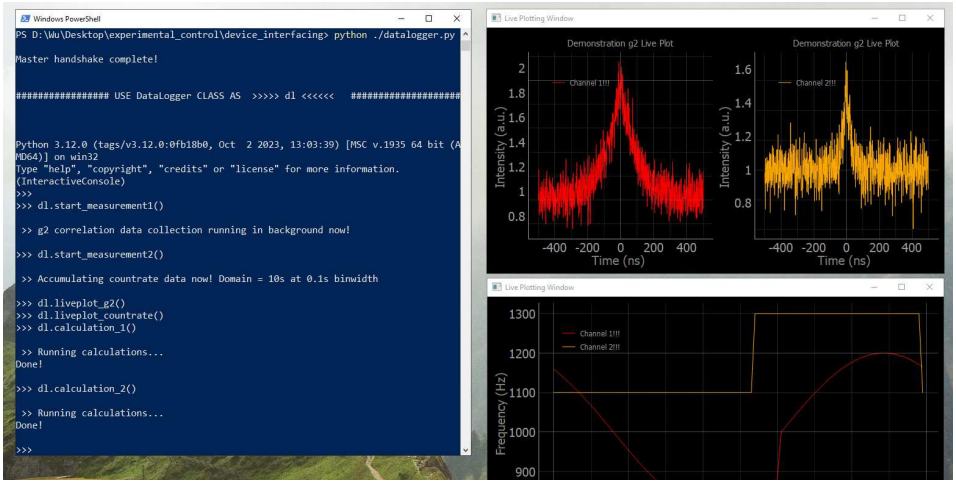

Figure 5: Desktop demonstration of using liveplotter as a module within a Python interactive console. As seen in the window on the left, the user may call functions from a "Datalogger" object to begin data collection, before opening asynchronous and interactive live plotting windows displaying the respective live data. The user may continue using the command line interface for the experiment while the live plots run in the background.

A popular plotting tool for data visualisation in Python amongst physicists would be Matplotlib. It is a powerful package that offers virtually all types of static scientific plots with attractive styling choices. Indeed, Matplotlib's animated plot functions were implemented for the first version of liveplotter. However, these functions quickly proved to be severely limited in refresh rate and scalability with data size. In contrast, the PyQtGraph package excels in live plots by using efficient data array manipulation sub-packages and PyQt5 (a Python GUI package) to handle graphics rendering in both two and three dimensions. While this alternative is clearly optimised for live plots, some obstacles were faced during development:

1. Early forms of the module required users to instantiate liveplotter for each live plot window and within separate threads. This allowed asynchronous plots that were fast and interactive, but each module instance was in fact a PyQt5 GUI application in itself. The result was significant overheads in the module back-end and only a maximum of three live windows could be run together on a standard desktop without significantly slowing down other processes.

2. The solution to this was to leverage available GUI methods provided by PyQt5. We now instantiate live plot windows as simpler *QWidget* objects that are in turn managed by a unified *QApplication* instance. While this design was more scalable and resource-friendly, it undermined asynchronicity as QApplications were designed to be run in the *main thread* of the program [The], which prevented liveplotter from being a non-blocking and drop-in module within a CLI.

To avoid switching entirely to an event-driven programming design (such as waiting for periodic time-outs to perform actions) usually used for PyQt5 GUIs but not QMLab's software,

liveplotter utilised the multiprocess package in Python to enclose all graphical methods within a sub-process. As shown in Figure 4, LivePlotProcess is the object running the QApplication within a sub-process. A sub-module in the main process, LivePlotAgent, then acts as an intermediary between the main instrument orchestration program (called Datalogger in this example) and LivePlotProcess, exchanging instructions and data in a thread-safe manner using inter-process queue methods. PyQt methods traditionally meant for a full GUI front-end interface, such as the "worker" QThreads that regularly update the live windows (QWidgets), were contained in sub-classes within the module.

With this mode of operation, the Datalogger, which represents the CLI experimental control session and handles the various instrument interfaces, may continue to be used to acquire and process data on-the-fly while live plots run in the background using the optimised PyQtGraph back-end. The Datalogger simply updates variables periodically with new data and live plot windows correspondingly update their visualisations. Figure 5 depicts a typical experimental control session in which a Datalogger is instantiated in a Windows Powershell console: liveplotter becomes an object instantiated by Datalogger and is used to generate several types of live visualisations, such as timing correlation histograms and rolling windows of photon count rates.

4 Useful Considerations for Software Design and Maintenance

We have presented two distinct examples in experimental physics where considerable effort was spent on custom and lab-specific software solutions tackling specific problems. These examples demonstrate how bottom-up software development in physics could look like, a field in which software development is accorded lower rigour. This section hopes to draw parallels between our experiences building our software and point out a few coding principles that were beneficial to us in keeping our code user-friendly and sustainable.

4.1 Abstraction

One of the key principles we realised early on was to ensure that the code was accessible and easy to use. Thus, *abstraction* was a natural way to achieve this [CS07]. Conceptually, this amounts to encapsulating complex operations behind a simple and consistent interface, which users can use as a "black box". Users can easily see the functionalities available, but not what the back-end might look like. While the contrary is possible, the focus here lies in *using* the software instead of *understanding* the software to an excessive degree, which may only be useful when debugging the software. Since only the relevant parameters need to be adjusted by the user, this has allowed us to make our software more accessible to our colleagues.

This also proved especially useful in the nanosquared project, as we have seen in Subsection 3.1 (nanosquared). Abstraction allows us to "hide" inelegant workarounds and the inevitable messiness of experimental setups, enhancing both (re)usability and maintainability of the codebase. In other words, abstraction facilitates the deployment of devices that may have very different control technologies but are conceptually the same in functionality.

It was also highly rewarding to reach out to device manufacturers for help. In the process of building code to interface with the beam profilers, they provided valuable example code and ex-

pertise of their products despite the age of the product model. We thus encourage our colleagues to reach out to external experts/vendors as much as possible to help reduce development time.

4.2 Modularity

Although closely associated, *modularity* and *abstraction* represent distinct concepts: the overarching goal of *modularity* is the separation of concerns, and this can come in many forms [McC03].

Python, being a multi-paradigm language, embraces different forms of modularity, supporting functional programming, procedural abstraction, and object-oriented design. However, Python's core remains deeply object-oriented [Fou24a], and the way it handles packages reflects that. Consequently, we found that the most natural way for us to achieve modularity was through the use of classes and objects to separate code. Since experiments usually involve multiple instruments working together, mapping *physical equipment* to *virtual objects* in code is a convenient technique when programming experimental control software.

As Figure 6 demonstrates, clearly delineated modules representing distinct entities create a *mix-and-match system* that greatly simplifies code composition. With lower-level hardware-interaction code boxed up and abstracted away, higher-level composed scripts using multiple modules also become more readable and intuitive. These reusable individual components form the centrepiece behind complex instrument orchestration systems and increase code reusability and longevity. Indeed, nanosquared (cf. Subsection 3.1) was conceptually divided into two groups of modules: hardware-interfacing objects and virtual, pure-software objects such as data fitting functions. Within each group, there are individual modules that use inheritance to create a structure that allows extensibility [Sun23].

This extensibility can be very useful as a concept for an experimental setup. For example,

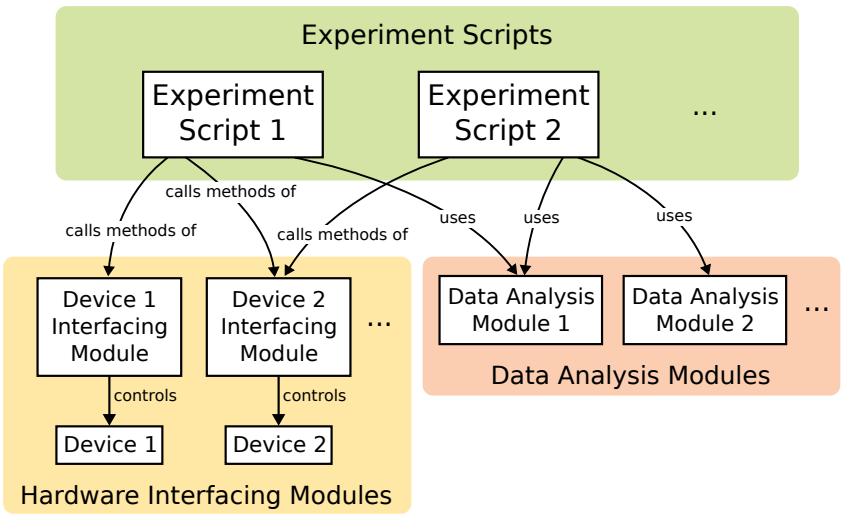

Figure 6: Modularity enables a *mix-and-match* system and encourages code reuse. Clear modules representing distinct entities greatly simplify code composition.

an experiment might use two different types of oscilloscopes A and B, but since they are both oscilloscopes, one would likely obtain the same type of information from them. This is when one would use a parent *oscilloscope* class that provides the data acquisition methods, and two child classes *A* and *B* that contain device-specific code (implementations) for the actual acquisition of the data from the device. From our experience, we found that this form of class *inheritance* to be particularly effective for factoring out common methods amongst a set of classes and provide hierarchical order to your code [SH15].

Effective modularisation (with abstraction) in this form is an excellent exercise of continuously grouping similar modules. This helps build new modules based on existing classes and makes the software easier to conceptualise for physicists by reflecting the layout of experimental setups.

4.3 Front/Back End Separation

While GUI-based systems have traditionally excelled in providing real-time data visualization and intuitive control, they are relatively inflexible in functionality and design, often being uniquely configured to an experimental setup. We observed that the problem is severely worsened when the instrument orchestration ecosystem becomes fundamentally GUI-oriented, where back-end functionalities and protocols are mixed with GUI properties and functions. This design has a high barrier of entry because it requires researchers to be fluent in building Python classes *and* using GUI toolkits (such as PyQT) to maintain and debug the code. This situation is detrimental to extending the functionalities of the ecosystem since new modules cannot be tested in experiments without the labour of integrating them into the main GUI first.

Hence, we strongly recommend making a clear front/back end separation, where the GUI is just a top-level layer that reads inputs and calls functions from the otherwise independent back-end script that can also be run via a CLI, such as Windows Powershell or Conda. In the example of QMLab, it was not obvious how this could be done before liveplotter was built as a drop-in module. By achieving this separation, we were able to focus on core instrument interface functionalities without being constrained by GUI implementation details. This allowed all group members (of varying programming knowledge) to take more ownership of software maintenance and create an adaptable experimental control system.

4.4 Version Control

A software tool we found crucial was version control of the entire development process. Good version control may be summarised as:

1. Tracking code changes and attributing them to corresponding authors.
2. Methodically handle conflicting or merging changes between different versions.

The first feature is analogous to how experimentalists keep laboratory notebooks to maintain an immutable record of each iteration of the experiment: The ability to refer or revert to a previous state of the experiment is vital for scientific rigour and reproducibility of results. The second feature enables organised collaboration on software and distributes workload amongst colleagues when developing software.

An industry-standard distributed version control system would be the **Git** tool, which offers decentralised editing amongst multiple collaborators but still easily manages and merges all changes. In addition, the internet enables us to perform *remote* version control, where code changes are archived online in *repositories*, and vast online communities may collaborate remotely. Some commercially available hosts online using Git include but are not limited to services like GitHub[5], Bitbucket[6] and GitLab[7]. Many institutions also self-host some form of Git server, often an instance of GitLab[8]. Despite Git's relative ease of use, consistent version control may still be the most daunting aspect of developing a software ecosystem for less experienced programmers. We recommend Chapters 15 and 16 in [SH15] for a step-by-step guide on using Git and GitHub. In general, adopting a well-established version control system within a research group facilitates shared code stewardship and promotes collaboration between colleagues, thus improving code sustainability.

4.5 Documentation

Researchers not specialised in programming often dread documentation because of how time-consuming the process can be on top of standard experimental record-keeping. Nevertheless, simple documentation already goes a long way to improve the maintainability and thus longevity of custom software. We find that as setups increasingly rely on advanced instrument orchestration to obtain useful results, allowing a new user to quickly understand the software directly benefits the progress of the experimental work. In Python, **type hinting** and **documentation strings** (docstrings) have been essential components of documentation for our software. The former was introduced in Python 3.5 [Pyt24] and explicitly shows the object type of variables and arguments for functions, thus making complex scripts more readable and easier to debug before run-time. The latter provides a standardised way of describing how functions work, their experimental relevance, and if needed the source literature from which a protocol was derived.

Additionally, a convenient tool for both creating high-quality documentation and alleviating the laborious nature of this task would be Artificial Intelligence (AI) code completion tools such as Github Copilot. These AI tools integrate directly into popular code editors (such as Visual Studio Code) and suggest code snippets and documentation as the user works [MMN+23]. By prompting the tool to interpret and automate documentation during and after writing the code, we can greatly improve the consistency and efficiency of our documentation. This was especially helpful in the early stages of our software when we did not yet have a standard format of documentation amongst all our modules. Obviously, such AI models are not completely accurate, but it is far easier to correct documentation suggestions than that for functions. Additionally, while the AI could infer the general intent of a function or class, we note that it is still important to supplement the suggested documentation template with corrections or meaningful information on less obvious aspects of the code.

[5] https://github.com
[6] https://bitbucket.org
[7] https://gitlab.com
[8] For example, Helmholtz Cloud provides a GitLab instance https://codebase.helmholtz.cloud for everyone in the *Deutsches Forschungsnetz* (DFN)

5 Conclusion

In this contribution, we examined the circumstances of building instrument orchestration in experimental physics to contextualise the difficulty in establishing an open-source research software standard across the community. We believe the disparity in programming literacy amongst experimental physicists and the inadequate recognition of good code in academia are major factors hindering a convergence in how we create our software. We acknowledge the bottom-up nature of software development in experimental physics, and simply encourage our colleagues to maximise code sustainability within their research groups by applying the presented concepts where possible. More than stylistic appeal, a well-maintained group repository makes research work FAIR and reproducible, and we hope that the ideas presented in this contribution will inspire better code. A more advanced guide that extensively covers the fundamentals of using software in physics research would be *"Effective Computation in Physics: Field Guide to Research with Python"* written by Anthony Scopatz and Kathryn D. Huff [SH15].

References

[ACCR19a] D. Allan, T. Caswell, S. Campbell, M. Rakitin. Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management. *Synchrotron Radiation News* 32(3):19–22, 2019.
doi:10.1080/08940886.2019.1608121
https://doi.org/10.1080/08940886.2019.1608121

[ACCR19b] D. Allan, T. Caswell, S. Campbell, M. Rakitin. Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management. *Synchrotron Radiation News* 32(3):19–22, 2019.
doi:10.1080/08940886.2019.1608121

[BCK+22] M. Barker, N. P. Chue Hong, D. S. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez, T. Honeyman. Introducing the FAIR Principles for research software. *Scientific Data* 9(1):622, Oct. 2022.
doi:10.1038/s41597-022-01710-x

[CG20] N. Caraban Gonzalez. Women in STEM. Nov 2020. General Photo. Specifically: https://cds.cern.ch/images/CERN-PHOTO-202011-160-74 © Noemi Caraban / CERN.
https://cds.cern.ch/record/2746101

[CER25] Trigger and Data Acquisition System, ATLAS Experiment at CERN. 2025.
https://atlas.cern/Discover/Detector/Trigger-DAQ

[CS07] T. Colburn, G. Shute. Abstraction in computer science. *Minds Mach. (Dordr.)* 17(2):169–184, Aug. 2007.

[Fou24a] P. S. Foundation. General Python FAQ — docs.python.org. https://docs.python.org/3/faq/general.html#what-is-python, 2024. [Accessed 18-10-2024].

[Fou24b] S. Fouquin. Photographs from UNIS Air-Ice-Sea Interaction AGF-211 Field Work. Private Communication, Apr. 2024.

[GFJ+04] C. Gaspar, B. Franek, R. Jacobsson, B. Jost, S. Morlini, N. Neufeld, P. Vannerem. An integrated experiment control system, architecture, and benefits: the LHCb approach. *IEEE Transactions on Nuclear Science* 51(3):513–520, 2004. doi:10.1109/TNS.2004.828878

[Han] S. Hannahs. LabVIEW Version Compatibility Chart Rev2022 - LabVIEW Version Compatibility. Accessed: 27.05.2024. https://info-labview.org/LabVIEW%20Version%20Compatibility.pdf

[HBC+22] S. Hettrick, R. Bast, S. Crouch, C. Wyatt, O. Philippe, A. Botzki, J. Carver, I. Cosden, F. D'Andrea, A. Dasgupta, W. Godoy, A. Gonzalez-Beltran, U. Hamster, S. Henwood, P. Holmvall, S. Janosch, T. Lestang, N. May, J. Philips, N. Poonawala-Lohani, P. Richmond, M. Sinha, F. Thiery, B. Werkhoven, Q. Zhang. International RSE Survey 2022. Aug. 2022. doi:10.5281/zenodo.7015772

[Het14] S. Hettrick. It's impossible to conduct research without software, say 7 out of 10 UK researchers. Software Sustainability Institute Blog, Dec. 2014. https://www.software.ac.uk/blog/its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers

[Het16] S. Hettrick. A not-so-brief history of Research Software Engineers. Aug. 2016. https://www.software.ac.uk/blog/not-so-brief-history-research-software-engineers-0

[Het18] S. Hettrick. softwaresaved/software_in_research_survey_2014: Software in research survey. Feb. 2018. doi:10.5281/zenodo.1183562

[Inc24] D. Inc. WinCamD-IR-BB — store.dataray.com. https://store.dataray.com/all-products/beam-profiling-cameras/wincamd-ir-bb/, 2024. [Accessed 18-10-2024].

[ISO21] ISO 11146-1:2021. Lasers and laser-related equipment – Test methods for laser beam widths, divergence angles and beam propagation ratios. Standard, International Organization for Standardization, Geneva, CH, July 2021.

[Jtv15] J. L. Johnson, H. tom Wörden, K. van Wijk. PLACE: An Open-Source Python Package for Laboratory Automation, Control, and Experimentation. *SLAS Technology* 20(1):10–16, 2015. doi:https://doi.org/10.1177/2211068214553022

[Jä24] Jäger GmbH. ADwin-Pro II – flexible and modular. Feb. 2024. Accessed: 27.05.2024.
https://www.adwin.de/us/produkte/proII.html

[KCAC20] L. J. Koerner, T. A. Caswell, D. B. Allan, S. I. Campbell. A Python Instrument Control and Data Acquisition Suite for Reproducible Research. *IEEE Transactions on Instrumentation and Measurement* 69(4):1698–1707, 2020.
doi:10.1109/TIM.2019.2914711

[Kod20] J. Kodosky. LabVIEW. *Proceedings of the ACM on Programming Languages* 4(HOPL):1–54, June 2020.
doi:10.1145/3386328

[Ltd24] O. O. S. Ltd. NanoScan (TM) 2s Pyro/9/5 Pyroelectric Scanning Slit Beam Profiler — ophiropt.com. https://www.ophiropt.com/en/f/nanoscan-2s-pyro-9-5-beam-profiler, 2024. [Accessed 18-10-2024].

[McC03] D. D. McCracken. *Modular programming*. P. 1183–1184. John Wiley and Sons Ltd., GBR, 2003.

[MLZ21] MLZ Garching. Software from Garching goes global. https://mlz-garching.de/englisch/news-und-press/news-articles/software-from-garching-goes-global.html, Dec. 2021. [Accessed 28-02-2025].

[MMN+23] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. M. J. Jiang. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203:111734, 2023.
doi:https://doi.org/10.1016/j.jss.2023.111734

[Nat24] National Instruments. LabVIEW Pricing Page. 2024. Accessed: 01.09.2024.
https://www.ni.com/en/shop/labview/select-edition.html

[NP24] J. Norris, N. Pawlowski. UNIS Air-Ice-Sea Interaction AGF-211 Field Report 2024. Private Communication, May 2024.

[Ols12] Olsen. The First 110 Years of Laboratory Automation: Technologies, Applications, and the Creative Scientist. *Journal of Laboratory Automation* 17(6):469–480, 2012. PMID: 22893633.
doi:10.1177/2211068212455631

[Pas] R. Paschotta. M^2 Factor. RP Photonics Encyclopedia. Available online at https://www.rp-photonics.com/m2_factor.html.
doi:10.61835/zxz

[Pyt24] Python Software Foundation. typing — Support for type hints. 2024. Python 3.12.0 documentation.
https://docs.python.org/3/library/typing.html

[RFG+22] F. Rahmanian, J. Flowers, D. Guevarra, M. Richter, M. Fichtner, P. Donnely, J. M. Gregoire, H. S. Stein. Enabling Modular Autonomous Feedback-Loops in Materials Science through Hierarchical Experimental Laboratory Automation and Orchestration. *Advanced Materials Interfaces* 9(8):2101987, 2022. doi:https://doi.org/10.1002/admi.202101987

[SCP99] SCPI Consortium. Standard Commands for Programmable Instruments(SCPI). May 1999. Accessed: 27.05.2024. https://www.keysight.com/us/en/assets/9921-01870/miscellaneous/SCPI-99.pdf

[SH15] A. Scopatz, K. D. Huff. *Effective Computation in Physics: Field Guide to Research with Python*. O'Reilly Media, 2015.

[Sun23] Y. Sun. Modularity in Software-Hardware Interaction for Experimental Physics, an Example. feb 2023. doi:10.5281/ZENODO.7677329

[The] The Qt Company Ltd. Threading Basics. Accessed: 01.09.2024. https://doc.qt.io/qt-6/thread-basics.html#simultaneous-access-to-data

[TZP21] A. Trenkwalder, M. Zaccanti, N. Poli. A flexible system-on-a-chip control hardware for atomic, molecular, and optical physics experiments. *Review of Scientific Instruments* 92(10), Oct. 2021. doi:10.1063/5.0058986

[VXI24] VXIplug&play Systems Alliance. The VISA Library Specification Revision 7.2.1. Jan. 2024. Accessed: 27.05.2024. https://www.ivifoundation.org/downloads/VISA/vpp43_2024-01-04.pdf

[Web21] S. J. Weber. PyMoDAQ: An open-source Python-based software for modular data acquisition. *Review of Scientific Instruments* 92(4):045104, Apr. 2021. doi:10.1063/5.0032116

[Wil16] M. D. e. a. Wilkinson. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* 3(1):160018, Mar. 2016. doi:10.1038/sdata.2016.18

[Wu24] M. Wu. Real-Time Data Visualisation in Experiments Using a Generalised Asynchronous Live Plotting Module, an Example in Python. Mar. 2024. doi:10.5281/zenodo.10805831