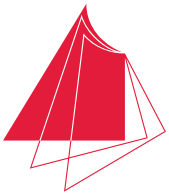


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Use Case oriented Performance Benchmarking of Python-based Graph Libraries

Bachelor Thesis of

Fabian Sorn

at the Faculty of Computer Science and Business Information Systems
Subject Area Distributed Systems (VSYS)

Reviewer:	Prof. Dr. rer. nat. Christian Zirpins
Second reviewer:	Prof. Dr.-Ing. Astrid Laubenheimer
Advisor:	Ivan Sinkarenko

10. November 2019 – 10. March 2020

Hochschule Karlsruhe Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik
Moltkestr. 30
76133 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Geneva; March 10th, 2020

. . . . . .

(Fabian Sorn)

Abstract

Good performance plays an essential role in the usability of software products. This is not only true for consumer software but also for professional software like monitoring GUI applications used at CERN, the European Organization for Nuclear Research. They allow experts to easily monitor a large number of devices and adapt their settings when necessary.

An essential component of such monitoring applications are graphs, which allow the presentation of a large data set in an easily understandable and compact form factor. Strong deviations and general trends can be discovered more easily which allows reacting to them. The work was conducted in the Applications section in the Beams Controls group. With the change to PyQt for new native GUI applications, the question raises, which python library is suitable for this task. Next to the features of each library, their speed in the later application plays a central role. A bad performance will significantly lower the productivity of the application's user since he is slowed down by having to wait for the system to respond to his interaction.

The goal of this work is to develop a framework, which allows testing the performance of python graph libraries in specific use cases. The user has to be possible to depict his use cases using the framework without much effort and without having to be familiar with performance testing. Using the framework should be easy for users familiar with other python testing frameworks. Through the on-demand creation of profiles for the given use case, advanced users should have the opportunity to find performance bottlenecks and eliminate them.

To reach this goal we begin with a general overview of the fundamental topics of data visualization and benchmarking as well as the application framework Qt. Following that, real use cases are presented which originate from monitoring applications, which are being developed at CERN at the time of writing. Based on these requirements and prior findings a concept for the central tasks of the framework as well as all involved components is developed. The concept is implemented using python 3 as the programming language of choice. The evaluation of the implementation is done in two separate steps. First, all collected use cases are implemented using the framework to make sure that the functionality meets our requirements. This is followed by the presentation of the recorded results for each use case. To evaluate the accuracy of the recorded results, a separate use case is compared to a minimal implementation of the fitting application.

Zusammenfassung

Eine gute Performance spielt eine wichtige Rolle, wenn es um die Benutzbarkeit von Software Anwendungen geht. Dies gilt im Konsumenten Bereich im selben Maße wie für professionelle Anwendungen, wie beispielsweise Monitoring Software am CERN, der europäischen Organisation für Kernforschung. Sie erlaubt es Experten, eine Vielzahl an Komponenten zu überwachen und deren Einstellungen, wenn nötig, anzupassen.

Eine essenzielle Komponente dieser Monitoring Anwendungen sind Graphen, die es erlauben, eine große Datenmenge gut verständlich und kompakt zu präsentieren. Starke Abweichungen oder generelle Trends in einem Datensatz können auf diese Weise schnell erkannt und entsprechend reagiert werden. Die Arbeit wird in der Applications Sektion der Beams Controls Gruppe erstellt. Mit dem Wechsel zu PyQt, als Framework für neu entwickelte, native Graphical User Interface (GUI) Anwendungen stellt sich die Frage, welche Python Graphen Bibliothek diese Aufgabe am besten erfüllt. Neben dem Angebot an Features spielt die Geschwindigkeit in der späteren Anwendung eine wichtige Rolle. Ist diese schlecht, wird Produktivität des Nutzers mit der Software maßgeblich geschmälert, da dieser durch lange Reaktionszeiten in seiner Tätigkeit ausgebremst wird.

Ziel dieser Arbeit ist die Konzeption und Entwicklung einer Lösung in Form eines Frameworks, die es erlaubt, mehrere Graphen Bibliotheken auf ihre Performanz in bestimmten Nutzungsszenarien hin zu untersuchen. Der Nutzer sollte dabei seine eigenen Szenarien ohne großen Aufwand frei abbilden können, ohne spezifische Kenntnisse über Performance Tests haben zu müssen. Die Nutzung sollte dabei vertraut für Nutzer anderer Python Test Frameworks sein und einen schnellen Einstieg erlauben. Durch auf Wunsch erstellte Profile soll es geübten Nutzern der jeweiligen Bibliothek ermöglicht werden, Flaschenhälse zu finden und diese bei Bedarf zu verbessern.

Um dieses Ziel zu erreichen, werden nach der Darstellung der grundlegenden Themenbereiche wie Datenvisualisierung, Benchmarking und dem Anwendungsframework Qt, Nutzungsszenarien präsentiert, die aus echten Monitoring Anwendungen herrühren, die während der Erstellung dieser Arbeit am CERN entwickelt werden. Basierend auf deren Ansprüchen und den zuvor gesammelten Information wird ein Konzept für die Funktionsweise zentraler Aufgaben und den Aufbau involvierter Komponenten entwickelt. Die Implementierung dieses Konzeptes erfolgt in der Programmiersprache Python in der Version 3. Die Überprüfung der Implementierung erfolgt in zwei Schritten. Zu Beginn werden alle gesammelten Nutzungsszenarien mithilfe des Frameworks implementiert und ausgeführt, um sicherzustellen, dass die Implementierung den Ansprüchen genügt. Darauf folgt eine Präsentation der erzielten Ergebnisse. Zur Evaluierung der Genauigkeit der gemessenen Ergebnisse, wird ein weiteres Szenario mit einer Implementierung in Form einer echten minimalen Anwendung verglichen.

Contents

List of Figures

List of Tables

1. Introduction

The following chapter will provide an introduction to this work. First, the setting in which the work is done will be described, starting with the organization followed by the team. Afterward, the fundamental problem and the goal of this work will be explained, rounded up by an overview of the structure of the following chapters.

1.1. Motivation

The big advantage that the raise of computing brought with it, was, that complex mathematical tasks, could be performed in very little time. Since the beginnings, a lot has happened and computers got much more powerful. Even with these improvements, the question of good performance could not be more relevant as today. We have to make sure, that the hardware and our algorithms are fast enough, to keep up with the tasks we want to accomplish. This demand is especially relevant in the scientific world, where often gigantic data sets have to be filtered, recorded and analyzed. Popular tools for such work are software products, that allow us to visualize data as graphs, since visualization allows us to have a much deeper insight into the data we want to understand. As with any type of software, the performance of graphs has to keep up with our high demands. One of the places, where this couldn't become more clear, is CERN, where the fundamental question the following work is based on, was researched.

1.2. CERN

The European Organization for Nuclear Research (CERN) is one of the biggest and most well-known research facilities in the world. It is most known as the host of one of the world's most complex and astonishing machines, the Large Hadron Collider (LHC) as well as the birthplace of the World Wide Web (WWW), on which we rely on daily everywhere around the world. [11] These and many more achievements and projects all contribute to the central mission at CERN: Finding out, what our universe is made of. To find answers to this question, CERN brings together over 17500 people from all over the world, to work together in many different fields including physics, engineering, computer science and more. Today, CERN counts 23 member states that collaborate on decisions made in the organization every day [18].

CERN's roots can be traced back to the 1940s, when a hand full of scientists saw the needs for Europe to advance its role in the scientific world by hosting an own research facility for physics. Starting with 12 original member states, CERN originally was founded based on this vision in 1954 located at the franco-swiss border, as the *Conseil Européen*

pour la Recherche Nucléaire, leading to today's well recognized acronym CERN. Until today, these member states are contributing to CERN's financing, organization and foundations to achieve its goals to expand the boundaries of human knowledge. [14]

1.3. Beams Controls Applications

CERN's main focus for research is particle physics. To continuously improve our knowledge in this field, CERN is operating the world's most powerful particle accelerator called LHC. The LHC allows us to gain a much deeper insight into the subatomic structure of the world around us bringing us closer to understanding the inner workings of our universe. CERN itself is divided into different departments thath have their own purposes. The Beams Department (BE) is responsible for developing software and hardware instruments for the accelerator complex. [9]

The LHC's task is to produce and accelerate two beams of charged particles, traveling in opposite directions. To achieve this,it is constructed as two circular pipes containing a vacuum, which are surrounded by magnets. The magnetic field created by these magnets can accelerate and steer particles passing by. If a beam of particles is injected into the accelerator, the strength of the magnetic field is increased with every round the beam travels in the accelerator, until the beam reaches speeds very close to the speed of light. Is this level of energy reached, the next step is to make particles from the two beams collide with each other. CERN is operating four experiments, Atlas, CMS, Alice and LHCb, where the particles can be led to collisions. The particle detectors than can record the results of the collisions in great detail for later analysis. The operation of the LHC is under one roof, the CCC. [16, 22]

Particle accelerators are set up from many different components, ranging from power converters that provide energy to the magnets to instruments responsible for monitoring all metrics describing the state of the beam. To operate all these different parts, a control system is necessary, which allows operators to change settings of components and monitor the resulting behavior. The development of this control system for the accelerator complex is done by the Controls Group (BE-CO) which is part of BE. [21] The control system itself is composed of different components that work together. Responsible for these components are different sections within the controls group. This work has been conducted in the Applications Section (BE-CO-APS), whose responsibility is, to provide software solutions for the many tasks of the control system. Part of these products are GUIs, that are vital tools for the operators' work. A GUI application allows monitoring the current state of the machine and react to occurring problems by altering the setting. To develop such monitoring applications, BE-CO-APS is providing different reusable GUI widgets, from which more complex monitoring applications can be developed.

1.4. Problem

A recent decision of BE-CO-APS at CERN was, to move from Java to Python for GUIs. This decision opens the door for many people, which aren't primarily software developers, to

write their GUI applications for their specific use cases. The framework of widgets which BE-CO-APS is providing, does also contain graph components, that allow operators to visualize data in their GUIs.

Choosing a library to implement graph widgets is not a trivial topic. Especially in python, there are many charting libraries, from which you can choose from, including matplotlib, Seaborn, PyQtGraph, Plotly, and more [52]. The comparison of features is in most cases not enough. Many users have specific needs and use cases, which rely heavily on the performance of the library. Compared to the evaluation of needed features and the offerings in libraries, evaluating the performance is not just a decision between *is available* and *is not available*. To answer the question, if a library is fast enough for a specific use case, we have to provide metrics, realistic use cases and a reliable way of testing the performance, that allow us to take a sound decision.

Benchmarking is a good way of answering such performance questions. Most known in these cases are benchmarks, which allow us to compare the speed of different hardware components, by running the same sequence operations on them and comparing the times, that they required to complete these tasks [59]. To compare the performance of different implementations of software, we can utilize a very similar approach. Instead of running the same code on different hardware, we can run different implementations of the same tasks on the same hardware and measure the performance of each. For more complex operations, like visualizing data, this inevitably raises the question, how we can implement such a measurement and what metrics describe the library's performance.

Benchmarking for different implementations would not only allow us to compare the same high-level operations between different libraries, but also the development of certain operations in the same library over time. For the user, such a benchmarking possibility would also make a decision between libraries much easier, since he could actively test his demands and use cases on different libraries to find the library that fits his performance needs the best.

1.5. Planned Solution

The goal of our work is, to develop a benchmarking framework for python graph libraries. The benchmark framework will be used to develop a suite of benchmarks, that can be used to verify the performance of a graph library written in Python in real-world use cases. The benchmark suite should not only give users comparable results to objectively judge performance but provide solutions to search and improve slow operations. To evaluate the benchmarks performance measurement capabilities, we will compare the results of the framework against an example production application.

1.6. Structure of this work

At the beginning of this work in chapter 2, we will create a common knowledge base that is necessary to understand the following chapters, by going through the basics of data visualization. To understand the technical decisions we took and implementations of our

solution, we will have a look at the technologies we will use. The following chapter 3 will deal with real use cases, that users of charting libraries at CERN have provided. From those, we will derive metrics, which we can use for our later implementation. The next chapter 4 then guides through the analysis and design of a benchmark framework, which allows us to run our graph library of choice against the collected use cases. Based on this design, we will implement the framework in chapter 5. In the following chapter 6 we will use the developed framework to implement the use cases from chapter 3 and record the results to see if the framework covers all of our requirements. To evaluate, how close the framework's results come to a production application, we will compare a use case with a minimal application with both displaying the same load. At the end of this work, chapter 7 will provide a summary as well as an outlook based on this work.

2. Fundamentals

This chapter will give an introduction to the fundamental topics that are relevant to the context of this work. In the first section, we will have a look at data visualization in general. We will start with a short exploration of the development in history leading to our modern usage of data visualization in the world of computing. Further, we will explore a model that describes, in general, the functionality and steps of modern data visualization frameworks and the different types of data visualization applications. Finally, the first section will present different usages of data visualization applications at CERN, which different scenarios exist and what solutions are offered at the moment from the BE-CO-APS section. In the second section we will move our focus on the topic of performance benchmarking and the different available types. Since the focus of our benchmarking framework will be GUI applications using Qt and its python bindings, we will have a look at its fundamental principles and functionalities in the third section.

2.1. Data Visualization

The following chapter gives a brief introduction into the history and development of data visualization, presents a model that describes modern data visualization frameworks and explores the usage of data visualization at CERN.

2.1.1. The development of Data Visualization

Good visualization helps us to understand patterns, trends and relationships in data much more easily and share discoveries with others. Visually representing data has its root in the earliest of humanity's history, where drawings on cave walls were a way to share stories, whose marks we can still find preserved until today. From there the idea of visualizing information, which we want to share, evolved into many different directions. One very early way to present data to viewers was to use tables. The oldest preserved documents presenting data in tables can be dated back to the 2nd century AD. Even though the information was still presented mainly in text, alignment, white spaces, and lines, were already utilized to support the viewer to navigate through data in large quantities. Visualizing data in a two-dimensional space using a system of coordinates has its roots in the 17th century. The two-dimensional cartesian coordinate system was the basis for displaying data as graphs. In the following 18th and 19th century more types of graphs like Bar Graphs and Pie Charts emerged, which we still use in modern representations of datasets today. [30]

Visualization does not only help us in understanding static data but also discovering trends in its development over some time. In 1753, Jacques Barbeu-Dubourg created a

graph displaying a large historical timeline, called the Carne Chronographique. The graph displays events from a total period of 6480 years, drawn by hand on a 54-foot long paper roll. Entries are listed chronologically behind each other from left to right. Special about this paper roll was, that it visualizes a big amount of historical data while keeping a small form factor, which is achieved, by rolling the large paper onto a scroll. Both parts of the scrolls were connected to keep always the same distance to each other, presenting the Viewer view on a part of the entire data. To move the timespan of this detailed view you could simply roll the paper further into one direction to go back or forward in history. [26]

Figure 2.1.: Jacques Barbeu-Dubourg's Scroll of History
Source: https://earlyamericanists.files.wordpress.com/2015/05/chronographie-universelle.jpg

With the invention of computers and the rise of more affordable models in homes and offices, data could be visualized much faster and easier than ever before. The labor-intensive process of drawing graphs by hand on special paper was replaced by computer-generated ones that could be created with the click of a button. This drastic improvement in ease of use further increased the popularity of data visualization and brought it into many more fields. [25]

One of these areas was the integration of interactive graphs into GUIs. No matter what language is chosen by a developer to write GUI applications, chances are high that there already exist powerful libraries for visualizing data. Compared to printed charts and graphs, graphs in software can be a much more powerful tool, since they do allow the user to interact with their data and alter its visual representation. Additionally, graphs displayed on a screen can be updated when new data is available compared to printed ones. A model that describes the functionality of such libraries, is the Data Visualization Pipeline. [31]

2.1.2. Data Visualization Pipeline

The Data Visualization Pipeline describes the transformation of raw data to a visual representation of this data which can be displayed on a screen. This transformation is achieved through a sequence of four operations that are executed to transform the data step by step.

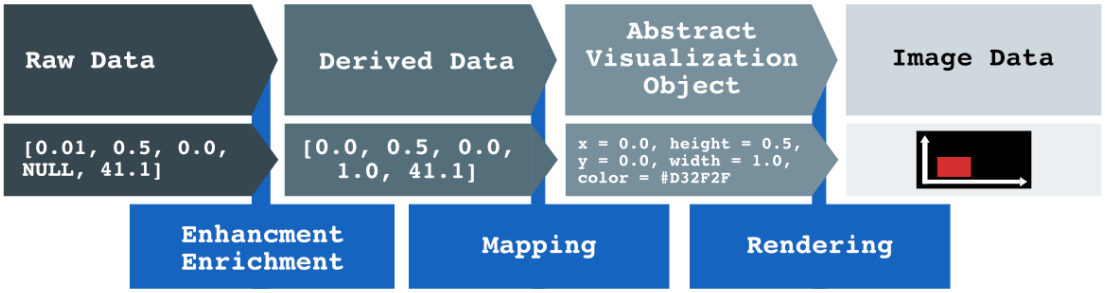


Figure 2.2.: Transformations and Resulting Data of the Data Visualization Pipeline

The starting point of the pipeline is the raw data acquired from a process we are collecting data from. At this point, the data has not yet been altered in any way. The first transformation we execute on this data is an operation to enhance or enrich the data. This can include interpolation for missing points in the dataset or smoothing filters for noise in our dataset we want to get rid of. In general, this step includes every calculation based on our raw dataset, which is useful but does not yet produce any visual information. The resulting dataset from this operation is called the derived data.

The next transformation in the Pipeline is the mapping of the derived data onto Abstract Visualization Objects (AVOs). These are not yet the image itself, but objects that describe the later visual representation of an entry in the dataset using certain attributes. An example of such an attribute could be a color, which is calculated from one of the dimensions in derived data. An example of this transformation step would be the mapping of a float value representing a measured temperature to a corresponding color representing a specific temperature range.

The last step in the pipeline is the Rendering, which produces an actual pixel image from the AVO. This step often involves operations from computer graphics like transformations between object, scene and device coordinates. With these transformations, it is possible to produce a two-dimensional pixel image of a three-dimensional scene. The rendering step, of course, depends highly on the technologies used and the AVOs we want to display. Compared to other usages of computer graphics, the goal of rendering for data visualization is not the production of a photo-realistic image, but the creation of an abstract approximation, that allows us to understand the scientific data it represents. Unnecessary details would in this case only harm the performance of the pipeline and distract the viewer. [31, 50]

Based on this model, there are three different types of visualization software systems. These different types of systems will bring different requirements to each step of the pipeline.

The first type is used in situations, where the source for the data is run once in the beginning, the data is collected and the visualization is done later. This type of software is mainly used for visualizing results of experiments running once without constant repetition. An example of such a system is the acquisition of data coming from a particle detector. The recorded collision is not something that will constantly be repeated over a long time, so the main priority of performance is, that the storage system can keep up with the high bandwidth data coming from the experiment. The visualization of such data is often done on heavily filtered versions, that only take data into account, which is interesting.

The second type of visualization software systems is runtime monitoring software. Compared to our first scenario, this software type requires all steps of the pipeline to be completed more than once. As soon as new data gets available from the source, it is passed through each step of the pipeline, to be visualized at the end. One way to take load from a very work-intensive step in the pipeline is to accumulate a certain amount of data before passing it onto this step.

The third type of visualization software allows the conductor of the source, to interactively influence its execution. This means, that a runtime monitoring system is extended using inputs, which allow the interaction between user and data source. If the user sees problems rising while monitoring the process, he can use the inputs provided by the software system to alter the input parameters. The success of the changes can be actively monitored while it is running. [31]

All of these three types of applications can be found to a certain extend at CERN. In the next section, we will have a look, how such software systems are used in a real scientific environment.

2.1.3. Charting at CERN

Visualization Software Systems of the first type can be found at CERN for example in the experiments using one of CERN's particle detectors. During Run 2, which lasted for four years and ended in October 2018, all four experiments produced around 25 Gigabytes of data per second. Even for the most modern storage solutions, storing such a band-width of data is an unrealistic challenge. This would require massive amounts of high-performance storage devices, which would be a prohibitively expensive solution. To reduce the amount of storage space needed, the Atlas Experiment filters out around 99 percent of data in a two-step filtering process and only leaves the most promising events. The actually stored data can then be analyzed and visualized offline. One example of such a display is a three-dimensional view of recorded collisions in the ATLAS Control room, which can be followed live online if the detector is running. [15, 12, 23, 24]

Implementations of the second and third types of data visualization software systems can be found as well at CERN, especially for monitoring continuously running processes. One of these places is the CCC, whose main purpose is the monitoring of the accelerator complex. Operators, experts for components of the complex, work here around the clock, to make sure that all of them are running as expected. The CCC is set up from four circular areas called *islands*, which are dedicated to monitoring different parts of the accelerator complex. These parts are the Technical Infrastructure (TI), Proton Synchrotron (PS), Super

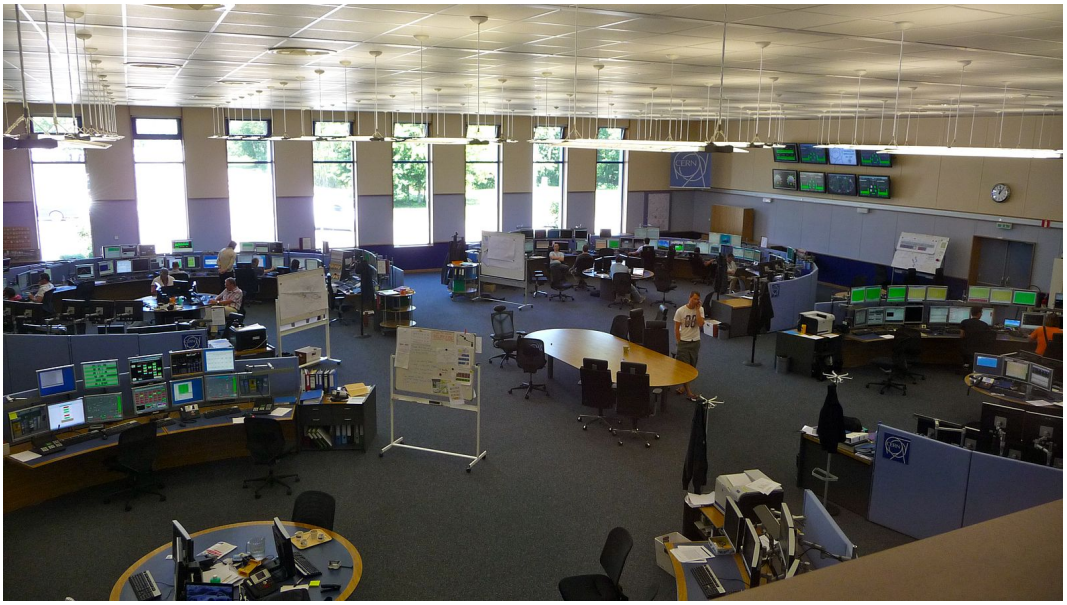


Figure 2.3.: CERN Control Center

Proton Synchrotron (SPS) and LHC. Collecting all of them under one roof allows them to directly keep contact with each other making communication between them more efficient. [7]

Consoles are computers in the CCC that allow interaction with the accelerator complexes control system through GUIs. Operator's Consoles are used for setting parameters of components in the control system. GUI applications running on these machines can be categorized as the inputs of data visualization software systems of the third type. Next to Operator Consoles, there are big wall-mounted displays called Fixed Displays or Vistars, which are running runtime monitoring applications in the form of dashboards, which allow operators to monitor the status of a specific component of the accelerator complex and see the resulting behavior of their interaction. Since they do not allow any direct interaction, Fixed Displays can be categorized as visualization applications of type two. [21]

One Vistar used as a reference for the feature development of a graph library for PyQt was the LEIR Vistar. LEIR is CERN's Low Energy Ion Ring, which is responsible for transforming a longer bunch of lead ions into shorter ones, preparing them for the injection into the LHC. Its Vistar contains a graph at the top displaying the injection of particles, the energy level, and the different cycles. [17]

2.1.4. JDataViewer

Fixed displays and GUI applications running on an Operator Console are built using components provided by BE-CO-APS. Previous to PyQt, BE-CO-APS focused mainly on Java and Swing to develop graphical user interfaces. JDataViewer is a powerful library that allowed users to easily create charts and interact with them. The library was developed in house at CERN since products on the market at that time were satisfying the charting

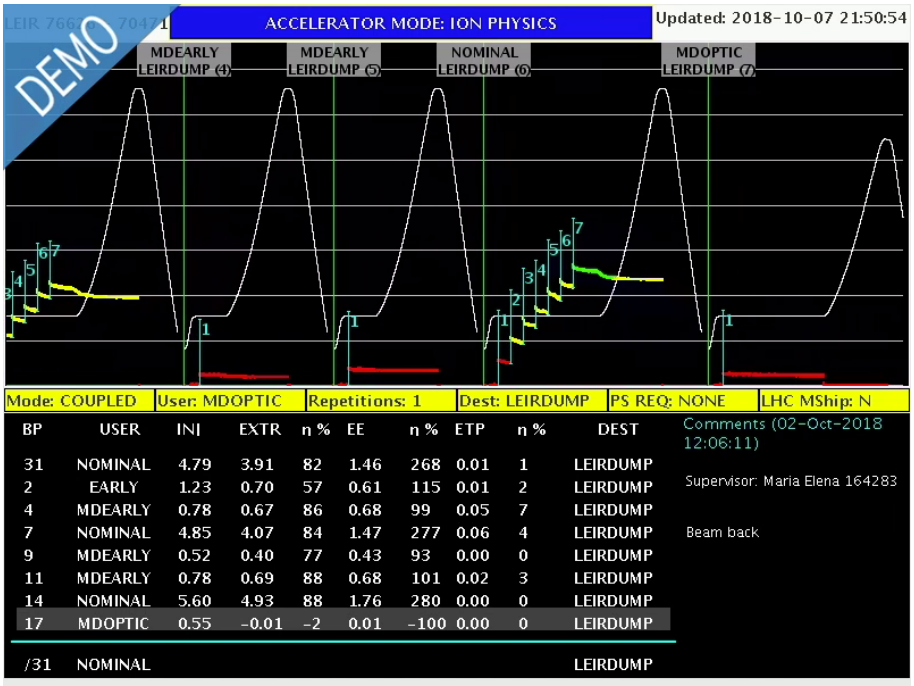


Figure 2.4.: LEIR Vistar in the CCC

needs of the users. One feature, which separates JDataViewer from the rest of libraries on the market, is its capabilities, to use charts for editing the underlying data set it is representing. Additionally to its large offerings in Features, JDataViewer provides class-leading performance when used in runtime monitoring applications. The goal for its PyQt predecessor is to offer similar features and performance. [33]

2.2. Benchmarking

The main advantage the invention of computers introduced into our lives was the massive increase in speed and the possibility to automate tasks, which would have taken us a time to complete in the past. Since then, computing speeds have increased drastically over time. According to Gordon E. Moore, co-founder of Intel, one of the world's largest semiconductor chip manufacturer, the transistor count in a dense integrated circuit doubles every second year. While this does not mean, that computing power doubles exactly in the same way, it shows us roughly, with which speeds computing power is increasing. Even with these developments, we still can easily find computing tasks, that can take modern computers a lot of time to complete, especially with the increase in the amount of data which we work with.

One way of handling highly complex computing tasks is to acquire more powerful hardware, which can process more quickly through the given data. This approach harbors a simple problem because the hardware we can acquire is often bound to a specific budget. Since more computing power will inevitably mean a higher count of transistor cores and clock-speeds, the price of hardware will raise with its computing power. A potentially less

costly approach is the optimization of the used software. By utilizing the power of our hardware more efficiently, we can achieve the same results with less powerful hardware. These two options lead to the question, which hardware and software combinations are performant enough to carry out the work we want. To answer this question, we need a procedure, which allows us to analyze performance objectively. Only objective results allow us to make a sound decision in which hardware and software combinations are suitable for our use cases. [45]

Benchmarking is a procedure that allows us to objectively compare the performance of a system or process, to find out, which workloads they are capable of handling. The usage of Benchmarking is not restricted to computing. In the field of Performance Management, benchmarking is a popular approach to evaluate the effectiveness of processes in a business. In journals testing new computing hardware, you can often find results, which tested hardware achieved in common benchmarking applications to give the potential customer a way to compare different offerings on the market. [3]

2.2.1. Synthetic Hardware Benchmarks

Hardware benchmarks can range from simple and synthetic sequences of operations to much more complex applications like benchmarks that try to test the performance of a component in a much more real-world test scenario.

Before synthetic benchmarking applications were a thing, manufacturers used Million Instructions per Second (MIPs) as a measurement to describe the performance of their chips. This would describe, how many machine instructions a chip could execute per second. Additionally Floating Point Operations per Second (FLOPS) were used to supplement the computing performance description in MIPs, since high numbers in MIPs were often achieved using less expensive integer operations. Until today, FLOPS can be found as a performance description for computing hardware, especially for Graphics Processing Units (GPUs). The GeForce RTX 2080, a to the time of writing very recent GPU from the manufacturer NVIDIA, can theoretically reach up to 10.6 TeraFLOPS (TFLOPS) or $10.6 * 10^{12}$ FLOPS. [8]

The first-ever program that was explicitly designed to test the performance of computer hardware was called Whetstone. The first version was published by H.J. Curnow and B.A. Wichmann in 1976 and was written in the programming language Algol 60. The goal of Whetstone was to test a hardware's performance without relying on any hardware-specific instructions. Instead of them, a collection of different higher-level operations containing integer arithmetic, floating pointer arithmetic, if statements, calls and more, which we still use frequently in modern programming languages. When executing the benchmark, all of these instructions are repeatedly executed. By using different weights for different operations, the results of the benchmark, in the end, could be described in a measurement called *Whetstone operations per second.* Another early benchmark that resulted from a library of linear algebra subroutines was LINPACK, which measured a computer's speed in solving operations on matrices. The result of the benchmarks were published in FLOPS. [59]

A more modern suite of benchmarking applications is provided by SPEC. Spec is a non-profit organization that offers a standardized suite of benchmarks, that can be used

to evaluate hardware performance. A new focus that SPEC brought up into the world of benchmarking, was the consideration of energy efficiency next to performance as a measurement of interest. [53]

Although it plays a very large role, the hardware is not the only factor that we have to take into consideration when talking about performance in computing. The second important part is the Software. No matter how fast the provided hardware is, if the running software is written without fully utilizing the hardware's capabilities, the resulting experience will not be as fast as it could be. Because of this, we can apply the same benchmarking efforts on software products as well.

2.2.2. Software Performance Tests

Next to benchmarking hardware, we can also measure the performance of the software. Testing performance is an important quality aspect in software development since it allows us to find performance-related problems. Especially in applications with multiple components involved, each of them can become a bottle-neck under certain circumstances. There are different variants of performance testing you can test your system against. The most common one is called Load or Stress testing and tests the application under different load scenarios, from very common to extreme ones. How these scenarios do look like depends on the type of application. For web applications, for example, a scenario could be a lot of concurrent users visiting the website. To see, if the system can withstand high loads for a longer time period, Endurance tests can be conducted. Spike Testing, on the other hand, involves switching very fast between low and high stress levels. Next to high load scenarios, other topics can be the focus of performance testing as well. This includes testing different configurations for the system, scalability with different hardware resources, different amounts of requested or written data and more.

Executing performance tests on a system allows us to compare them, determine if they will meet our performance requirements and help us finding components, whose performance needs to be improved. Since performance tests should always be conducted with a goal in mind, we have to ask ourselves, what this goal is for our particular system. Since our focus will lie solely on end-user GUI applications, server response times or amount of concurrent users are no suitable performance goals. Much more interesting for us are render response times, which describe, how much time passes when presenting the results of an operation to the user. How important this response time is for the user experience, shows a test conducted by Google, in which a page, whose load times were increased by half a second (from 0.4 to 0.9 seconds), lost 20 percent of its traffic. [49, 51, 2]

Another example of software benchmarks include testing the same task using different implementations under the same conditions. One example of such software benchmarking is the comparison of different programming languages. The free software project *The Computer Language Benchmark Game* uses a set of different algorithms to test the performance of different programming languages. There are no restrictions on how to implement the solution as long as they withstand a set of unit tests, that make sure the implementation is actually correct. A description of the different benchmarks, the different implementations and the results achieved in each implementation can be found on the project's official web page (https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html).

Even though the results can hint the performance capabilities of different languages, the project states, that it is important to see them in context. The performance of these mostly synthetic, very isolated operations is not capable of displaying the performance capability of the language in its entirety. [60]

2.2.3. Application Benchmarks

For most real-life use cases, measurements like FLOPS are not very helpful, when choosing hardware or software products, since those numbers only describe a very narrow field of use. The tasks whose performance we do care about, are very high-level and can be divided into countless subroutines. Application Benchmarks allow us, to investigate the performance in such high-level use cases. Very common use cases, whose success is very highly dependent on a good performance, often already have a big offering in application benchmarks available on the market. Especially in journalism about computer hardware, you can find a very widespread usage of application benchmarks to describe the performance of newly released hardware products and how they will perform compared to other models. In advertisements for hardware products, benchmarks are used as well to prove the gains in performance available from new developments. For hardware that is often used to play video games with, many benchmarks exist, which allow displaying scenes similar to real video games while evaluating the performance of the system doing so. Some video games like the open-world action game *Grand Theft Auto V* even offer a benchmarking mode, where common scenes from the game are depicted together with the current performance on the computer it is running on. Other benchmarks like *DeepBench* or *MLPerf* allow us to gain insights into the performance of hardware for machine learning tasks. Such results can help tremendously in the buying decision for new hardware components. [42, 58]

2.2.4. Collection of Benchmarking Criteria

No matter, which type of benchmark we have been looking at, they can all be described through common criteria.

1) Benchmarks should be written in high-level programming languages. This allows us to port them easily onto other hardware. It is often required to execute benchmarks on different architectures without changing the implementation between them.

2) To give us a realistic view of the performance of a machine, the Benchmark should also representative of the task we want to perform later on the hardware. Running rendering benchmarks on graphics hardware does maybe give us a comparable number when it comes to rendering capabilities, but does, for example, no help us if we are looking for graphics hardware to execute a certain machine learning task on it. The best benchmark would always be the user's actual application.

3) Another fundamental requirement of a benchmark is, that the benchmark should be easily measurable. Only if we can measure the benchmark easily and reliably, we can produce comparable numbers in the end.

4) The last requirement for benchmarks is, that they should be widely distributed. If benchmarking results are only available for one certain piece of hardware or software, but not for its competitors, we do not have any comparability between them.

2.3. The Qt Application Framework

To effectively develop benchmarking tools for python graph libraries, we have to understand the principles of the underlying GUI framework first, which in our case is the Qt application framework. This chapter will present an introduction to the core principles of it.

2.3.1. Widgets, Layouts and Widget Hierarchy

For the implementation of the Benchmarking Framework, we have to focus on a GUI Framework, in which our widgets will be embedded and tested. Since BE-CO-APS has decided for Qt as the framework of choice for GUI applications, we will take the same decision for our implementation. This chapter will give a general introduction to the creation of GUI applications with the help of Qt.

Qt is a C++ Framework for developing cross-platform applications. It is currently maintained by the Qt Company and offers licenses for commercial and open source usage. Qt is not just a collection of GUI components, but a complex and powerful application framework. It is divided into three main packages:

QtCore offers non GUI related Core functionalities for applications.

QtGui offers GUI related data wrapper classes and utility functions.

QtWidgets offers a collection of high level GUI widget and layout classes.

`QtWidgets.QApplication` is the central application class in Qt. Only one instance of it can exist at a time, which represents the current state of our Application, but does not yet represent any visual components like windows. If we want to present a window to the user, one option is to use `QtWidgets.QMainWindow`. One way to create a window containing our GUI components in it is to subclass `QMainWindow`. In this subclass we will have access to all functionality of `QMainWindow` and its subclasses. One of these functions is `QtWidgets.QWidget.show()`, which will draw the window on the display presenting it to the user. The last step of every Qt GUI Application is to call `QtWidgets.QApplication.exec()`, which will start the Main Event Loop of the Application. More information about this Event Loop and its role will be discussed in section 2.3.3.

Listing 2.1 combines these mentioned steps in a running Qt Application written in Python Code. The main window does not yet contain any widgets but has a custom window title set to it. A simple application like this does only depend on classes from the `QtWidgets` package.

```
1 import sys
2 from qtpy import QtWidgets
```

```
3
4
5  class MainWindow(QtWidgets.QMainWindow):
6
7      def __init__(self):
8          super().__init__()
9          self.setWindowTitle("Hello World!")
10         self.show()
11
12
13 if __name__ == "__main__":
14     app = QtWidgets.QApplication(sys.argv)
15     _ = MainWindow()
16     sys.exit(app.exec())
```

Listing 2.1: Hello World Qt GUI Application written in Python

Two additional details in regards to example 2.1 are worth mentioning. The first is, that we are passing of `sys.argv` to the `QApplication` constructor. This allows us to define parameters for the `QApplication` when invoking the python application from the command line. The second one is, that we use the return status from `app.exec()` when exiting Python. This allows us, for example, to mark our Python Application as failed, if the `QApplication` fails and exits with an exit code unequal to zero.

To add content to our window, Qt offers the concept of widgets. The base class for every widget is `QtWidgets.QWidget`. Next to `QWidget`, the package `QtWidgets` offers many more ready to use widgets for many different purposes. All common User Interface (UI) building blocks, like labels, checkboxes, combo boxes and more can be found in it ready to be added to a window. `QWidget` itself can also be used as a wrapper around multiple other widgets. Each widget can be given a parent on creation. Using this functionality, the user can define a tree-like parent-child hierarchy for all widgets starting from the main window. A big advantage that this hierarchy offers, is, that with the deletion of a widget, all its child widgets will be deleted as well.

Next to widget classes, Qt offers layout classes to order multiple children inside a parent widget. Some very basic layout options include:

`QtWidgets.QVBoxLayout` aligns widgets vertically below each other.

`QtWidgets.QHBoxLayout` aligns widgets horizontally next to each other other.

`QtWidgets.QGridLayout` aligns widgets in a grid.

Next to those basic ones, Qt also offers more complex layouts with additional functionality. Other options for window layouts are grouping widgets using Container Widgets. As an example, the class `QtWidgets.QTabWidget` allows grouping widgets into different views, which can be switched between by clicking on different tabs. By utilizing these different widgets and layout options, you can build more complex window layouts as seen visible in figure 2.5. On the right side of the depiction, you can see the widget and layout types contained in the window visible on the left. The source code for this window is appended in A.1. [44]

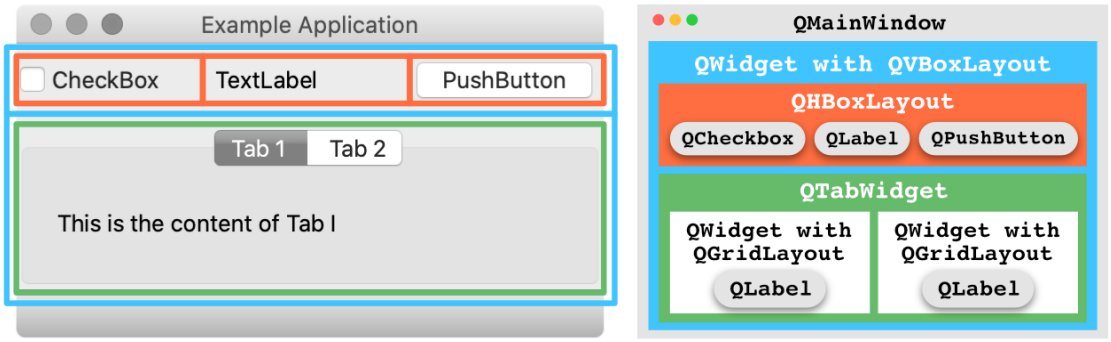


Figure 2.5.: Layouts and Widgets in a Qt Application Window

2.3.2. Signals and Slots

At this point, we have learned, how we can define the appearance of a GUI application, but interacting with these components does not yet invoke any logic. For this, we have to define a connection between user interface components and a function, which implements the logic we want to invoke. Many GUI Frameworks realize this connection through callbacks. A callback is a pointer to a function, which is invoked when interaction with a certain user interface element is detected. One problem this approach harbors is, that with callbacks it can't be ensured, that the passed arguments have the right type.

To solve this problem, Qt offers a concept called Signals and Slots for communication between objects. All widgets from the package `QtWidgets` do inherit from `QtCore.QObject` and do offer different standard signals and slots for informing about and reacting to state changes. The usage of Signals and Slots is not restricted to the communication between GUI elements and business logic but can be used in any classes derived from `QtCore.QObject` .

Signals are public functions, which emit messages informing about changes in the state of the object who emitted them. Slots are simply normal instance methods with the addition, that they can be connected to Signals to receive and react to the messages sent from this Signal. Since they are normal instance methods, Slots can also be called normally from code. The connection between both is set up, by connecting a signal of an object to a slot of an object. A signal can be connected to multiple slots and a slot can be connected to multiples signals. Depiction 2.6 shows a schema of an example scenario where different objects are communicating using signals and slots.

One big advantage of Signals and Slots is, that objects are fully independent of each other, since neither the slots know, which signals it is connected to, nor the signals know which slots it is connected with. Compared to callbacks, the Signal and Slots have more Overhead, which makes their execution slower. Compared to any for example list operation requiring `new` or `delete` , this overhead is still much smaller. In real-life applications, the performance losses coming through the usage of Signals and Slots are insignificant. [39, 44]

Listing 2.2 shows a simple scenario, where the signal of a `QtWidgets.QLineEdit` , is connected to a slot of a `QtWidgets.QLabel` . Both signals and slots are already imple-

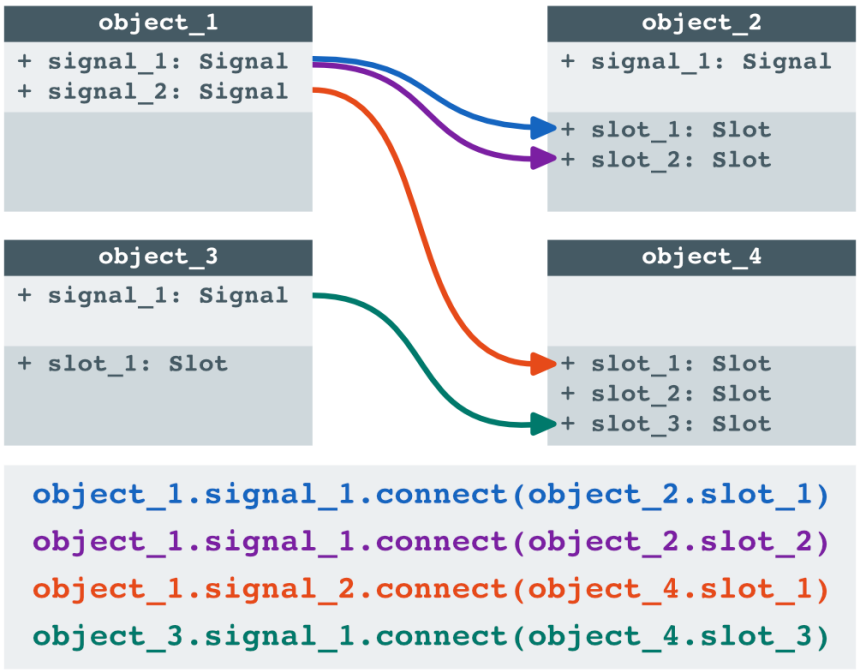


Figure 2.6.: Communication between objects through Signal and Slot Connections

mented in these classes and can be simply connected. If you change the text in the line edit widget, the signal `textChanged` will emit this new text, which the Slot `setText` receives and displays.

```
import sys
from qtpy import QtWidgets


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args):
        super().__init__(*args)
        self.label = QtWidgets.QLabel()
        self.edit = QtWidgets.QLineEdit()
        # Connect Text-Changed-Signal to the Set-Text-Slot
        self.edit.textChanged.connect(self.label.setText)
        self.setup_layout()
        self.show()

    def setup_layout(self):
        """Take the widgets and put them into a layout"""
        central_widget = QtWidgets.QWidget()
        central_layout = QtWidgets.QVBoxLayout()
        central_widget.setLayout(central_layout)
        self.setCentralWidget(central_widget)
        central_layout.addWidget(self.label)
```

```
        central_layout.addWidget(self.edit)
        self.setWindowTitle("Signals and Slots Demo")


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    _ = MainWindow()
    sys.exit(app.exec())
```

Listing 2.2: Connecting items through singals and slots

In some cases, for example, when we create own widgets, we have to define own signals and slots. To do so, we extend our example with another label type `WatchLabel` derived from `QtWidgets.QLabel`, which offers a slot `set_time`, which takes a float value, interprets it as a timestamp and displays it. To update this label, we will introduce another class `WatchTick` derived from `QtCore.QObject`, which emits a signal 60 times per second containing the current timestamp. Listing 2.3 shows the definition of these two components. Since we have typing information in the signal and the slot definition, Qt makes sure, that both will work together. If both types do not match, a `TypeError` will be raised.

```
import time
from qtpy import QtWidgets, QtCore, QtGui


class WatchLabel(QtWidgets.QLabel):

    def __init__(self):
        QtWidgets.QLabel.__init__(self)
        font = QtGui.QFont("Courier")
        font.setWeight(QtGui.QFont.ExtraBold)
        self.setFont(font)

    @QtCore.Slot(float)
    def set_time(self, timestamp: float):
        """Takes timestamp and displays it in a readable format."""
        self.setText(str(time.ctime(timestamp)))


class WatchTick(QtCore.QObject):

    sig_time = QtCore.Signal(float)

    def __init__(self):
        QtCore.QObject.__init__(self)
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.emit_time)

    def emit_time(self):
        self.sig_time.emit(time.time())

```

```
31      def start(self):
32          self.timer.start(1000/60)
```

Listing 2.3: Defining a custom signal and slot

Similar to listing 2.2, we can now connect the new slot with the signal, which is depicted in listing 2.4. After starting the timer, the signal will be emitted around 60 times per second. This will update the label, which then displays the current time.

```
1           self.watch_tick = WatchTick()
2           self.watch = WatchLabel()
3           # Update the watch label as soon as the ticker sends an update.
4           self.watch_tick.sig_time.connect(self.watch.set_time)
5           self.watch_tick.start()
```

Listing 2.4: Using custom Signals and Slots

Since Qt Applications are event-driven, they can be created as a single-threaded application without any problems. In more complex applications, some operations can take a bit of time to complete. When executed in the GUI Thread, this would block all interaction with the UI. To keep the GUI responsive, work-intensive operations can be moved to a separate Thread for parallel execution using, for example, Qt's `QtCore.QThread`. Signals and Slots offer a very convenient way to communicate between these multiple Threads. An example of this would be, that the GUI Thread spawns a new Worker Thread, in which a computing-intensive process is running. If finished, this worker thread can inform the GUI Thread about its completion by emitting a signal. The GUI thread can then display the results from the worker thread.

When working with multiples Threads in a Qt application, it is important to know about the different connection types which Qt offers. These connection types control, when and in which Thread the connected slot is executed. A `QtCore.Qt::DirectConnection` for example leads to the Slot being executed directly in the Thread of the emitting object. Between the Signal being emitted and the Slot being called, the control is not given back to the event loop. A `QtCore.Qt::QueuedConnection` on the other hand executes the Slot in the Thread where the receiving object lives. With this type, the control is returned to the event loop. The default connection type `QtCore.Qt::AutoConnection` will decide between queued and direct connection depending on in which Thread the emitting and receiving objects are living. [37]

2.3.3. The Event System

Qt Applications can very often be written without using parallel programming at all, which vastly decreases the complexity of the application and increases maintainability. Qt is an event-driven toolkit, which means, that applications are designed to wait for events and react to them appropriately. In general, Events can be described as anything that happened, which the application has to know about. These Events can be Mouse Clicks, Timer Timeouts or internal events which are supposed to tell a part of the GUI to redraw. Events can come from different places: Mouse Click or Key Events are most likely coming from the Window Manager while other events can come either from the application code

or the Qt Framework. Qt does not necessarily handle events immediately as they arrive, but queues them for later handling. The Qt Event Loop is responsible for iterating over these queued Events posted to the Event Queue to process them accordingly.

If we have a look at the examples in section 2.3.1, we will see the statement `app.exec()` in them. This call will start Qt's Event Loop and will be blocking until the application terminates. The code placed below this line will only be evaluated when the application has already quit. This means, that operations for showing a window or adding widgets to it will have to be placed above it. After the main event loop is started, it will retrieve events from the queue and process them one by one. Prior defined operations for showing a window or adding contents to it are such Events. In Qt, all these events are represented by a class derived from `QtCore.QEvent`. A `QtGui.QMouseEvent` for example is derived from it and extends it with functions specific to this type of Event like `QtGui.QMouseEvent::buttons()`, which returns information about buttons that were pressed during the mouse click event.

To make sure that Events are properly processed, they have to be published to the Event Loop and the Event Queue. While such events often come from outside the application code, they can also be created and published within it. To create an Event, an instance of the fitting Event Type has to be created. As mentioned, for standard Event types, Qt offers already implemented classes. Events can be published using either `QtCore.QtCoreApplication.sendEvent()` or `QtCore.QtCoreApplication.postEvent()`.

The first will lead to immediate delivery and handling of the Event without any involvement of the Event Loop, while the later one will add the Event to the Event Queue, from where it will be delivered by Qt when it is its turn. Listing 2.5 shows as an example the creation of a Mouse Press Event for a Click with the Left Mouse Button and its propagation to a widget contained in the window. The widget which receives the Event is a widget derived from `QtWidgets.QLabel`, which does not yet react in any way to mouse clicks.

```
from qtpy import QtCore, QtGui, QtWidgets

class ChangingLabelOne(QtWidgets.QLabel):
    pass

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self):
        super().__init__()
        widget = ChangingLabelOne("Test")
        self.setCentralWidget(widget)
        event = QtGui.QMouseEvent(
            QtCore.QEvent.MouseButtonPress,
            QtCore.QPoint(0.0, 0.0),
            QtCore.QPoint(0.0, 0.0),
            QtCore.QPoint(0.0, 0.0),
            QtCore.Qt.LeftButton,
            QtCore.Qt.LeftButton,
            QtCore.Qt.NoModifier
        )
        QtCore.QCoreApplication.postEvent(widget, event)
```

```
22         self.show()
```

Listing 2.5: Positing a MousePressEvent to a Widget

When it is time to process an Event, Qt is responsible for delivering events to the fitting widgets. The type of the Event does influence in which way it is delivered. If an event is not accepted by a child widget, it is propagated to its parent widget, which then has to opportunity to accept it. Widgets have two options when defining their behavior on the delivery of Events. The first option is `QtCore.QObject.event()`, which is the general event handler for all types of events. Here it is possible to intersect events of a specific type. For all other events your widget does not care about, the base classes implementation of `event()` can be called. Listing 2.6 shows the implementation of a label class derived from `QtWidgets.QLabel`, which accepts Mouse Click Events. If the widget detects a Mouse Click Event, it accepts it and increases its font size by one point.

```
1  from qtpy import QtWidgets, QtCore, QtGui
2
3  class ChangingLabelOne(QtWidgets.QLabel):
4
5      def event(self, e: QtCore.QEvent) -> bool:
6          """Change Label Font Size thorugh Mouse Click"""
7          if e.type() == QtCore.QEvent.MouseButtonPress:
8              font: QtGui.QFont = self.font()
9              if e.button() == QtCore.Qt.RightButton:
10                 font.setPointSize(font.pointSize() - 1)
11             elif e.button() == QtCore.Qt.LeftButton:
12                 font.setPointSize(font.pointSize() + 1)
13             self.setFont(font)
14             return True
15         else:
16             return super().event(e)
```

Listing 2.6: Change Font Size through Mouse Press using General Event Handler

If the widget is only supposed to handle specific types of Events, more specific Event Handlers can be overwritten. Listing 2.7 shows the same Use Case as in listing 2.6 but achieved through overwriting the specific event handler dedicated to Mouse Button Clicks.

```
1  from qtpy import QtWidgets, QtCore, QtGui
2
3  class ChangingLabelTwo(QtWidgets.QLabel):
4
5      def mousePressEvent(self, e: QtGui.QMouseEvent):
6          """Change Label Font Size thorugh Mouse Click"""
7          font: QtGui.QFont = self.font()
8          if e.button() == QtCore.Qt.RightButton:
9              font.setPointSize(font.pointSize() - 1)
10         elif e.button() == QtCore.Qt.LeftButton:
11             font.setPointSize(font.pointSize() + 1)
12         self.setFont(font)
```

Listing 2.7: Change Font Size through Mouse Click using Mouse Press Event Handler

Which events are delivered to which widget, is decided by Qt, where different Factors can play a key role. For the Mouse Click Events, the position of the pointer at the time of the Event does play a key role. Qt also offers the possibility to install global Event Filters which allow us to filter out events we do not like our Widgets to receive. These Filters simply are classes derived from `QtCore.QObject`, which implement the protected method `QtCore.QObject.eventFilter()`. This Event Filter can then be installed on any `QtCore.QObject` and is working for it and its children. To install it globally for the whole application, it can be installed on the `QWidgets.QApplication` instance as you can see as an example in listing 2.8. As before, we have to return `True` in the event filter to signalize Qt, that the event does not have to be propagated further in the widget hierarchy.

```
import sys
from qtpy import QtCore, QtGui, QtWidgets

class MouseClickFilter(QtCore.QObject):

    def eventFilter(self, _: QtCore.QObject, e: QtCore.QEvent):
        """Filter out all Mouse Clicks, single or double"""
        intercept = [QtCore.QEvent.MouseButtonDblClick,
                     QtCore.QEvent.MouseButtonPress]
        if e.type() in intercept:
            print("Filter Mouse Click.")
            return True
        return super().eventFilter(_, e)

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    app.installEventFilter(MouseClickFilter(parent=app))
    win = QtWidgets.QMainWindow()
    win.show()
    sys.exit(app.exec())
```

Listing 2.8: Installing an application wide event filter for single and double mouse button presses

Listing A.2 contains a runnable application that demonstrates these three concepts of event creation, event handling and event filtering in a single simple application window similar to the here shown code snippets. It contains two labels reacting to mouse clicks by changing their font size and a button sending a mouse press event to both labels. The checkbox contained in the window installs mouse click event-filter on both labels and removes it again when unchecked. [40, 4, 57]

2.3.4. Python bindings

Even though we do now have a basic understanding of the fundamental principle of the Qt Framework, we have not yet mentioned, how we can use a C++ application Framework in Python Code. This is possible through a principle known as Python bindings and is not exclusive to the Qt Application Framework. Python Bindings allow you to write code in Python using already existing C and C++ libraries, which are already well-received and

popular. One very popular Python Binding for the Qt Application Framework is PyQt, developed by Riverbank Computing Limited. Riverbank provides different versions of PyQt for different versions of Qt, for Qt5 we will pick PyQt5, which is the most recent major version of PyQt. [34]

Qt is a big framework containing over one thousand classes, which have to be accessible through Python to reflect the complete set of features. While it is generally possible to write these bindings by hand, it is a lot of work. To automate the creation of Python Bindings, Riverbank has developed the tool SIP, which can generate Python bindings from interface header files with similar syntax to C++ header files. These files define what classes and methods should be exported as well as the python module it should be exported into. Additionally, it is possible to define the translation between Python and C++ Objects. SIP takes these header files and generates C++ code, which can then be compiled to an extension module for the Python Interpreter. These Extension Modules allow us to import and call C and C++ libraries within our Python Code, which in the Case of PyQt is the underlying Qt framework. [35, 47, 27]

Working with a C++ library comes with a few caveats which we will have to keep in mind especially when working with the Qt Framework from Python. One particular caveat, which can easily lead to errors, is, that Objects do exist not only as Python objects but as C++ objects as well, which the Python object is referencing. Qt has an own Garbage collection mechanism, which can delete objects if their parent object is deleted or if they do not have any parent object. The deletion on the C++ side does not influence the existence of the object on the Python side in any way. This makes it possible to access a deleted C++ object through their Python references, which will raise Runtime Errors. When trying to delete a widget, it is also not enough to delete the Python object with `del self.my_widget`, since the C++ object underneath can exist further. Not being aware of this can easily create applications with severe memory leakage, especially when creating and deleting repeatedly new widgets, for example when opening new dialogs in an application. To free the memory occupied by a widget's C++ object, we have to tell Qt to delete the object and its children, as seen in the listing 2.9.

```
self.my_widget = QWidget(parent=other_widget)
# Posts an event to Qt's Event Loop to delete the object
self.my_widget.deleteLater()
# optionally we can delete the python referece now
del self.my_widget
```

Listing 2.9: Example for properly deleting a QWidget

Another caveat when working with PyQt appears when working with multi-threaded applications. When working with `QtCore.QThread` on the C++ side, execution can be truly parallel on multi-core systems. Working with different threads in Python is possible as well, but if a `QtCore.QThread` runs CPU bound Python Code, it won't be executed in parallel no matter how many CPU cores are available. The reason for this is Python's Global Interpreter Lock (GIL). The GIL was introduced to python to keep track of object references in multi-threaded code. Before a thread accesses any Python objects, it has to hold this lock, which means, that code running inside the GIL can't take real advantage of multi-core systems. [1, 29]

Since PyQt5 is not the only available Python binding for Qt, we will use the Python package `qtpy` for all code using Qt's API. `qtpy` is a simple pure Python abstraction layer, that delegates all imports to the Qt bindings installed on the system. This has the advantage, that the code, that was developed with PyQt5, will run in environments which have the Qt binding PySide or PyQt4 installed. Listing 2.10 shows how using `qtpy` affects Qt imports.

```
# Direct usage of a Python binding, will fail if environment doesn't use PyQt5
from PyQt5 import QtCore, QtGui, QtWidgets
# Using qtpy for Qt related imports, will work with PyQt and PySide
from qtpy import QtCore, QtGui, QtWidgets
```

Listing 2.10: Qt imports using qtpy instead of a python library

3. Use Cases

This chapter will give an overview of different scenarios that we will use for the evaluation of the Plotting Libraries. All of these originate from teams at CERN, that are looking into PyQt as an option to implement different GUI applications, which also contain plots in them. Each of the three following subsection present one use case from Hardware and Timing (BE-CO-HT), Operations Group LHC (BE-OP-LHC) and BE-CO-APS.

3.1. Distributed Oscilloscope for BE-CO-HT

The first project interested in visualizing data using python graph libraries is the Distributed Oscilloscope developed in the BE-CO-HT section, which is responsible for the development, production, and support for the custom electronic modules used in the Control System. The goal of the Digital Oscilloscope is to synchronously monitor analog signals in a distributed system. To achieve this synchronization, the signals from various sources are time-stamped and sent to the GUI, where they can be displayed using graphs. The project is distributed into three architectural layers, which are depicted in figure 3.1.

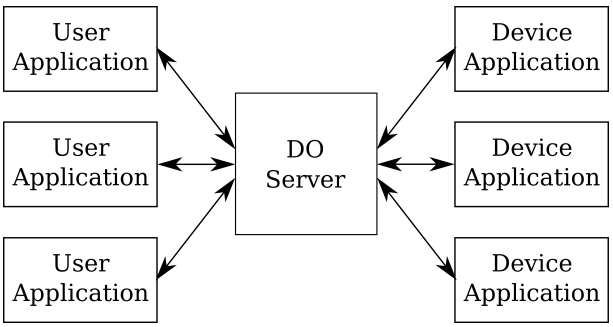


Figure 3.1.: Distributed Oscilloscope Architecture

The layer closest to the hardware the depicted signals originate from are the device applications, which provide access to hardware resources. The central layer is the Distributed Oscilloscope Server, which is responsible for managing all connections between device and user applications. The last layer closest to the users is the user applications. The application our use case originates from is a GUI application representing an oscilloscope, a device for displaying varying signal voltages over time. A screenshot of this application can be seen in figure 3.2. [41, 10]

For this GUI, BE-CO-HT is interested in displaying a plot in their application, which is showing up to eight curves with up to 100.000 points per curve. The gaol of this graph would be an update rate of 25 updates per second.

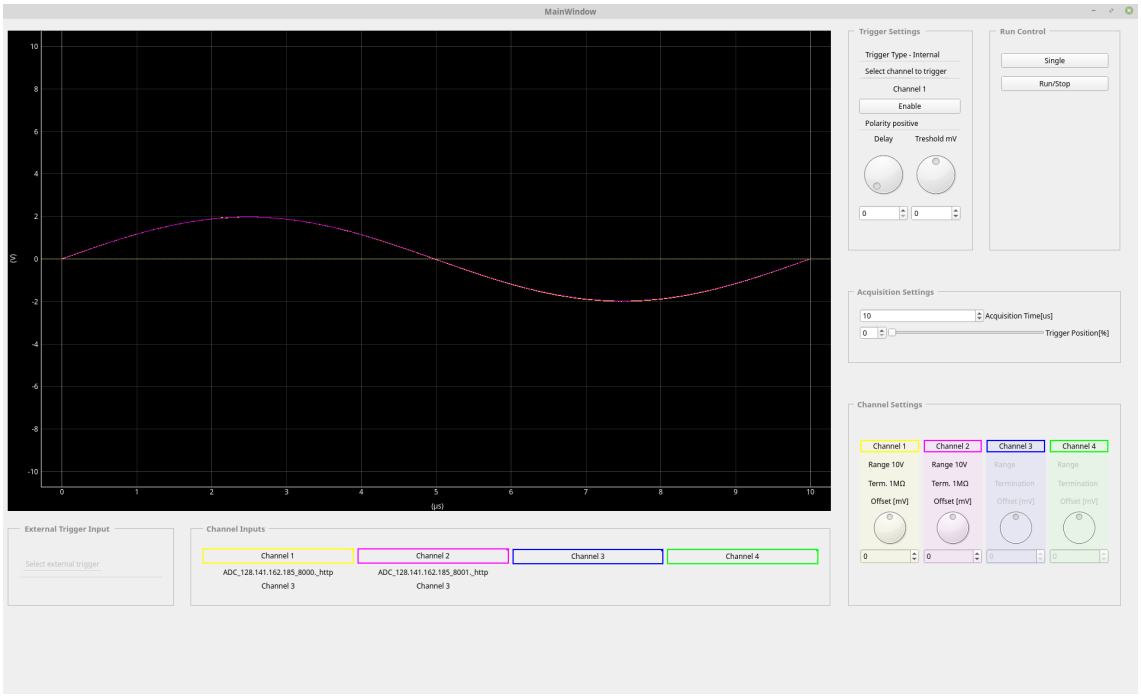


Figure 3.2.: Screenshot of the Distributed Oscilloscope GUI application

3.2. Monitoring application for BE-OP-LHC

The second Use Case we will investigate is coming from the BE-OP-LHC, who are interested in displaying a Line Graph containing 3000 datasets displayed as curves, who each will contain 2 * 3600 points. The data will be updated every second.

3.3. Linac4 Source GUI for BE-CO-APS

The third use case originates from BE-CO-APS. For this, multiple scatter plots should be displayed. The GUI Application will contain 4 different scatter plots, each containing up to 3 data sets, with each containing 1 hour of live data, which receives a new point roughly every 1.2 seconds. This results in 3000 visible points per data set.

The Application where this Use Case is originated from is a GUI for the Linear Accelerator Linac4, whose task it is to boost negative hydrogen ions to high energies. The GUI will allow monitoring and manipulation of different device settings. Image 3.3 shows a screenshot of an early version of the application with multiple scatter plots in the upper right part of the window. [13, 48]

3.4. Performance Metrics for User Interfaces

All of these Use Cases have in common, that they are displaying live data, which will be delivered with a certain frequency. This means that the graph has only a certain time

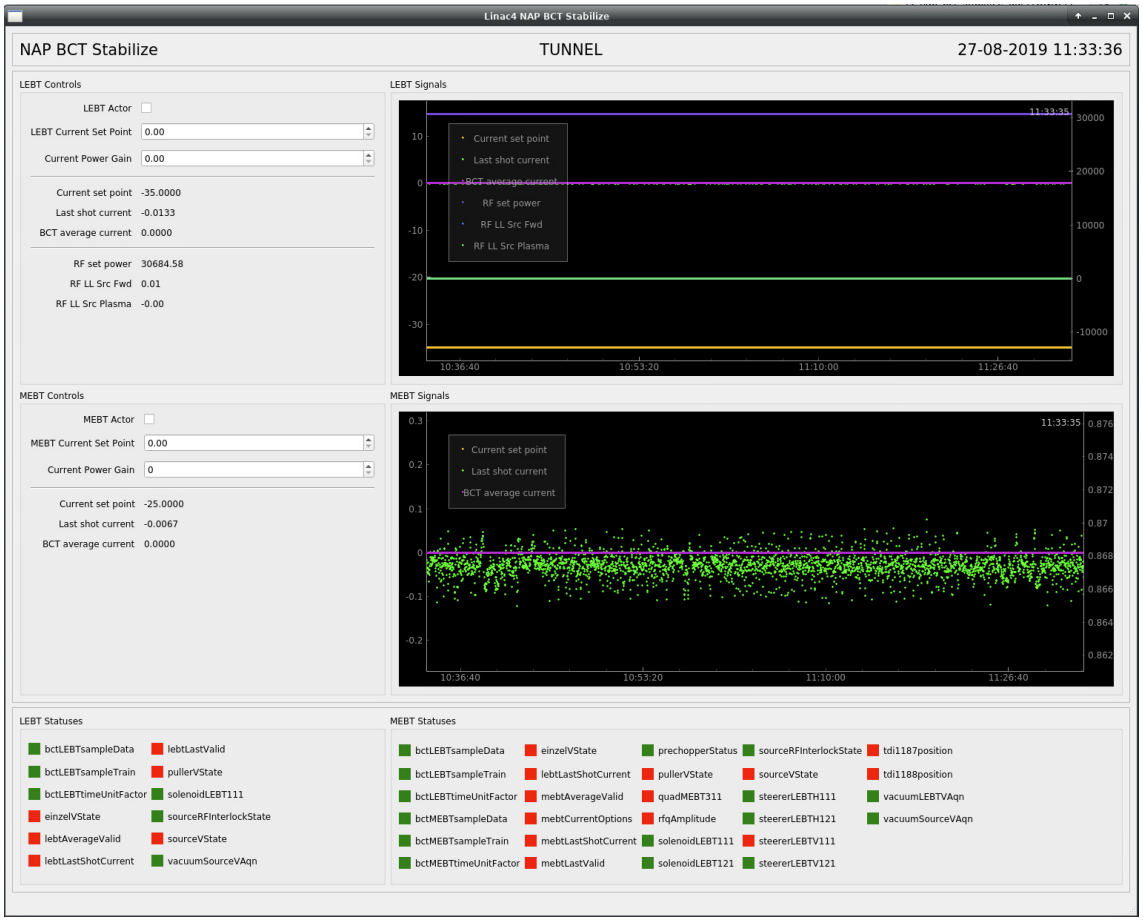


Figure 3.3.: Screenshot of the Linac4 Source GUI application

to redraw until the next bunch of data arrives. Even for applications containing graphs, which are not updated regularly, the time it takes the graph to redraw is very important for the user interaction. Slow redraw times will lead to stuttering user interfaces. As a result, our highest priority when investigating the performance of plotting libraries should be the redraw speed of the graph.

From a Usability Engineering perspective, rendering performance is a vital aspect, since it greatly impacts GUIs response times. The general boundary, in which a system feels like it is responding instantly to a user's interaction is 100 milliseconds. Delays over one second can already interrupt the thought process of the user since the delay is getting very noticeable. Delays over ten seconds require feedback from the system to signal the user it is performing long-running tasks. [46]

An often used measurement for describing a system's rendering performance is the frame rate, which describes the frequency, with which images appear on a screen. This size originates from the movie industry, where the standard frame rate is 24 Frames per Second. The minimum frequency needed for the human eye to see a consecutive movement from a sequence of images is as low as 16 Frames per Second. This does not mean, that frame

rates beyond this won't be recognized by the human eye. Increasing the Frame Rate leads to the reduction of motion blur, increasing details and clarity of a moving image.

Especially in the area of computer games, high frame rates are desirable, since they will not only enhance the visual experience but give the player a strategic advantage in a competitive environment. A popular demonstration of the increased clarity of high frame rates is the Blur Buster's UFO Motion Blur Test website, which shows the same scene of a fast-moving cartoon UFO in different frame rates (https://www.testufo.com).

In many Hardware Benchmarks, frame rates can be found as the central description of a hardware component's abilities to render a complex scene. To measure this, the hardware benchmark will try to rerender the scene as fast as possible. At the same step in the rendering process of a single frame or image, the current timestamp is recorded. From these series of timestamps, we can calculate the difference between two adjacent timestamps. This concept is known as Delta Timing and gives us information, how long the rendering process is taking us per Frame. Next to a performance description, delta timing can also have other usages. In Video Games, for example, it allows us to find out, how far an object in the displayed scene should have moved since the last displayed frame. As a result, figure movement stays consistent, even with different frame rates under different load scenarios. [6]

From the recorded timing information, we can calculate the frame rate f from a set of m recorded time stamps t_n as follows. [54]

$$f = \frac{1}{\frac{\sum_{n=1}^{m-1} t_{n+1} - t_n}{m-1}}$$

While frame rates already help us describing the performance, they do not yet help improving it. One way of finding operations in software worth optimizing is by profiling it. Profilers are software tools that can measure different aspects of performance, like execution time, memory management, garbage collection and more. The type of profiler we will focus on is deterministic CPU utilization profilers, which give us information about the time spent in functions, the callers of it and the number of calls to them. Next to finding slow-performing functions, they can uncover as well, if a function is called more often as intended by the author of a program. With this information, optimization efforts can be directed to where they really can make a difference. [19]

The standard option in Python for deterministic profiling is the C extension *cProfile*. Compared to the profiler *profile*, which itself is written in Python, cProfile offers a much lower overhead, which makes it suitable for profiling longer running programs. To provide collect information about the execution of a program, cProfile uses hooks for events, which are offered by the Python interpreter. These hooks are callbacks, which allow cProfile to receive and record timing information about every function executed. For profile and cProfile, there are two limitations, which you have to be aware of. The first one is the granularity of the underlying timer used for recording the duration of an operation. On most systems, this is around 0.001 seconds. The other limitation is the lag between the event being dispatched and the profiler getting the current time. Since profiling is introducing a certain overhead to the execution of a python program, it is not suitable for benchmarking itself. [28]

4. Analysis and Design of a Benchmark Framework

In this chapter, we will further analyze the requirements of a benchmarking framework, which allows users to benchmark specific charting operations. The results of this analysis will then be combined with the results from the previous two chapters for the design of our framework. First, we will have a closer look at the two libraries, PyQtGraph and Matplotlib, which we will use for the implementation of our use cases. Afterward, we will further analyze the requirements for the use cases we have to be able to depict in the framework. In the last section, we will then design the components as well as the main functionality of it.

4.1. Python Graph Libraries

When benchmarking different Python Graph libraries, we will focus our attention on two popular contenders for the implementation, which are very well accepted in the scientific community.

4.1.1. Matplotlib

Matplotlib is without a doubt the standard library for 2D data visualization in Python. It offers publication quality visualization as well as an interactive environment. The project was initialized by John D. Hunter as an easy to use Python 2D plotting library, especially for users familiar with Data Visualization in Matlab. [55, 32]

Matplotlib's central item is the `matplotlib.figure.Figure`. The Figure itself does not yet display anything, neither a plot with axes nor data. A plot is referred as an `matplotlib.axes.Axes`. A figure can have multiple Axes in it. For each dimension in the data space, the Axes contains `matplotlib.axis.Axis` objects, which represent the minimum and maximum data limit for each dimension of the data. An Axes object can be personalized through axis labels and a title. In Matplotlib terminology, Artists are everything that is visible in a Figure. This includes Labels, Lines and Bar Graphs, Axis Items and more. The last crucial component is the Canvas, which is not really a visual component in the plot, but the component responsible for rendering the image. A summary of all components can be seen in A.3

Matplotlib is built for many different use cases. While showing data in GUI applications is one of them, other use cases, like generating plots as images for publications are possible as well. This is achieved by Matplotlib's different backends. All available Backends can be divided into interactive and hard copy ones. An example of an interactive backend is

in our case PyQt5, but other Frameworks like Tkinter or PyGTK are supported as well. Non-interactive backends are for creating image files in different file formats like PNG, SVG, PDF and more. [56, 43]

Listing 4.1 shows the creation of a window containing a plot representing a sinus curve through a line, a scatter plot and a bar graph. For interaction, Matplotlib offers a toolbar, which lets you select between different interaction modes like panning and zooming. As a backend Matplotlib's Qt5 backend was used. The resulting window can be seen in Screenshot A.4.

```
import numpy as np
from matplotlib.backends.backend_qt5agg import (
    FigureCanvasQTAgg as FigureCanvas,
    NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

class MatplotlibWindow(QtWidgets.QMainWindow):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Generate the data we want to display
        x = np.array(range(100)) * 0.2
        y = np.sin(x)
        # Plot + Navigation Bar
        self._canvas = FigureCanvas(Figure())
        self._plot = self._canvas.figure.subplots()
        self._toolbar = NavigationToolbar(canvas=self._canvas, parent=self)
        # Add a curve, scatter plot and bar graph
        curve = self._plot.plot(x, y, "-")
        scatter = self._plot.plot(x, y + 2, "o")
        bars = self._plot.bar(x=x, height=y + 1, bottom=-3, width=0.15)
        # Setup Window Content
        layout = QtWidgets.QVBoxLayout()
        central_widget = QtWidgets.QWidget()
        central_widget.setLayout(layout)
        layout.addWidget(self._toolbar)
        layout.addWidget(self._canvas)
        self.setCentralWidget(central_widget)
```

Listing 4.1: Definition of a Window containing a plot created with Matplotlib

4.1.2. PyQtGraph

PyQtGraph is a pure python plotting library. Even though it does not offer as many features as other Python plotting libraries like Matplotlib, PyQtGraph promises much better performance. The project was initialized by Luke Campagnola and is focused on providing plotting functionalities for engineering and science applications. It provides simple plots containing line graphs, scatter plots, bar graphs and more, but also image and video displaying, Region of Interest widgets, 3D visualization and more. Since our

benchmarking efforts will be tightly focusing on the collected use cases, we will restrict our usage of its features mainly on two-dimensional plotting. PyQtGraph uses Qt's Graphics View for drawing, which is a Framework for fast visualization of a large number of custom 2D items. A big advantage it offers is a fast performance and the possibility to interact and transform the scene through operations like zooming or rotation. [36]

Since PyQtGraph is built on top of Qt features, integrating it into Qt applications is very simple. The central component for plotting is the `pyqtgraph.PlotWidget`, which is derived from `QtWidgets.QWidget`. When adding the plot widget to a window, it comes with different components on the inside, as seen in A.1. The central one is the `pyqtgraph.PlotItem`, which is the actual plot. The widget itself is only a wrapper for easy integration into Qt Applications. The plot item contains different components of the plot, including a `pyqtgraph.ViewBox`, the area data is visualized in, a `pyqtgraph.AxisItem` representing the View Range of the View Box, and a Title. Items that are actually representing a dataset are added to the View Box. For this, PyQtGraph is offering different types of representations like `pyqtgraph.PlotDataItem` for Scatter Plots and Line Graphs and `pyqtgraph.BarGraphItem` for representing data in a bar graphs.

Listing 4.2 shows the creation of a window containing a plot representing a sinus curve in different ways. It displays the same data sets in the same style as the Matplotlib example 4.1.1. The resulting window can be seen in A.2. [5]

```
import numpy as np
from qtpy import QtWidgets
import pyqtgraph

class PyQtGraphWindow(QtWidgets.QMainWindow):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Generate the data we want to display
        x = np.array(range(100)) * 0.2
        y = np.sin(x)
        # Plot
        self._plot = pyqtgraph.PlotWidget()
        # Add a curve, scatter plot and bar graph
        curve = pyqtgraph.PlotDataItem(x=x, y=y)
        scatter = pyqtgraph.PlotDataItem(x=x, y=y + 2, pen=None, symbol="o")
        bars = pyqtgraph.BarGraphItem(x=x, height=y + 1, y0=-3, width = 0.15)
        self._plot.addItem(curve)
        self._plot.addItem(scatter)
        self._plot.addItem(bars)
        # Setup window content
        self.setCentralWidget(self._plot)
        self.setWindowTitle("PyQtGraph Demo")
        self.show()
```

Listing 4.2: Definition of a window containing a plot created with PyQtGraph

4.2. Analysis

After having a closer look at the two plotting libraries, this chapter will focus on the analysis of the requirements which we will base the design and implementation of our framework on.

Users interested in the performance of graph libraries often have a very clear idea of what their use case looks like since they already know the background information of the data they want to plot. Because of this, the central interface to the framework should be these use cases. Most higher-level use cases can be broken down in a simple operation or a sequence of operations. From our collected use cases, we can define the following requirements.

Ease of Use The user should only define the relevant parts for his use cases. Everything else should be handled by the framework.

Widget and Operation The central items for defining a use case are a widget and an operation that is executed on this widget.

Multiple Use Cases Adding and removing use cases has to be possible without much effort.

Parameterization A use case has to be reusable for different parameters and has to be easily extensible.

Performance Expectations The author has to be able to define his performance expectations per use case.

Timeout In case a use case takes much longer than expected, it should be possible to define a timeout, which will terminate the use case's execution, if reached.

One framework type, which allows a very similar way of defining specific use cases, are testing frameworks like unittest or pytest for python projects. Since most users are to an extend familiar with their functionality, the interface should conform to the expectations built from the usage of these frameworks.

For easy execution of use cases, the framework will offer a command-line interface, which offers similar functionality as other python tools. The Command Line Interface (CLI) should conform to the user's expectations in the same way with a few additions specific to our needs.

Use Case Execution Similar to frameworks like pytest, it should be able to execute multiple use cases as well as single ones.

Meaningful Results After execution, the user should be able to see the results of his use cases. This includes the use case, its performance requirements, the achieved performance and the parameters used in the run.

Profiling Since Profiling introduces additional overhead to the code's execution, profiling should be optional and activatable through the CLI.

Profiling Visualization Since profiles often contain a large amount of data, they have to be visualized in a user-friendly way.

4.3. Design

This section will focus on describing the design of the framework. The outermost layer contains two components: the command-line interface and the business logic. The command line interface offers the user the possibility to start the execution of his benchmarking use cases. Additionally, it is responsible for presenting the results after the execution finished. The second component is the business logic, which is responsible for providing interfaces for defining use cases, the logic for executing them, as well as the functionality to record the achieved results. We will refer to the executing part of the framework as the launcher from now on.

4.3.1. Benchmark Execution

Figure 4.1 shows the communication between the user, the command line interface and the launcher when executing use cases from the command line. When the user starts the framework's executable, a CLI instance is created, which parses the command line arguments and instantiates the launcher. To keep the user interface exchangeable, launcher and CLI are strictly separated from each other. The launcher will import the passed use case modules, with which the initialization is completed. To start the execution, the launcher offers a `run()` method. Each collected use case will then be executed sequentially and its achieved results collected. After all use cases are executed and all results are available, they are passed back the CLI, where they can be presented to the user.

Next, we will have a more detailed look at the inner workings of the launcher. Its central task is the execution of use cases, which is visualized in depiction 4.2. Our goal is to allow executing a suite of use cases similar to testing frameworks allowing the execution of entire test suites at once. Each use case will have to be executed sequentially and in a separate environment, to prevent different use cases from having an influence on each other. The same principle applies to parameterized use cases. This means the execution will take place in a nested loop with the outer one cycling through the use cases and the inner one through its parameter combinations.

Visualization 4.2 introduces two new components, which build the execution environment for a single use case. The first component is the window, which will house our widget, on which we want to operate. The second component is the executor, which is responsible for executing the defined operation on the window and record timing information and profiling statistics. After our Launcher is initialized and our benchmark files are fully imported, we will execute each use case in all possible parameter combinations. For each combination, we create a new benchmarking window. The window will then be passed to the executor. The executor will access all use case related information and operations through this window reference. For starting the execution, the executor offers a `run()` function, which gets called by the launcher.

Depiction 4.3 shows the execution of a single use case in more detail. Before each operation, the current timestamp is taken. If a profile is supposed to be created, the profiler is started right after that. Next, the operation, which we want to benchmark, is executed on the passed window reference. After it and all of its side effects are completed, the profiler is stopped if necessary. Its statistics are added to the already collected ones. This

execution cycle is repeated until the defined repeat count or the timeout is reached. In the end, the timing information and the profiling statistics will be returned to the launcher.

Figure 4.4 shows a class diagram of all classes involved in the sequence diagrams and their relationships with each other. Additionally, it shows the use case interface with all its attributes and functions based on our analysis in section 4.2. To define a new use case, the user can subclass this class and implement all necessary components. The results of a single use case are represented by its own class, which can be passed between each of the components involved in the execution until they can be displayed by the CLI.

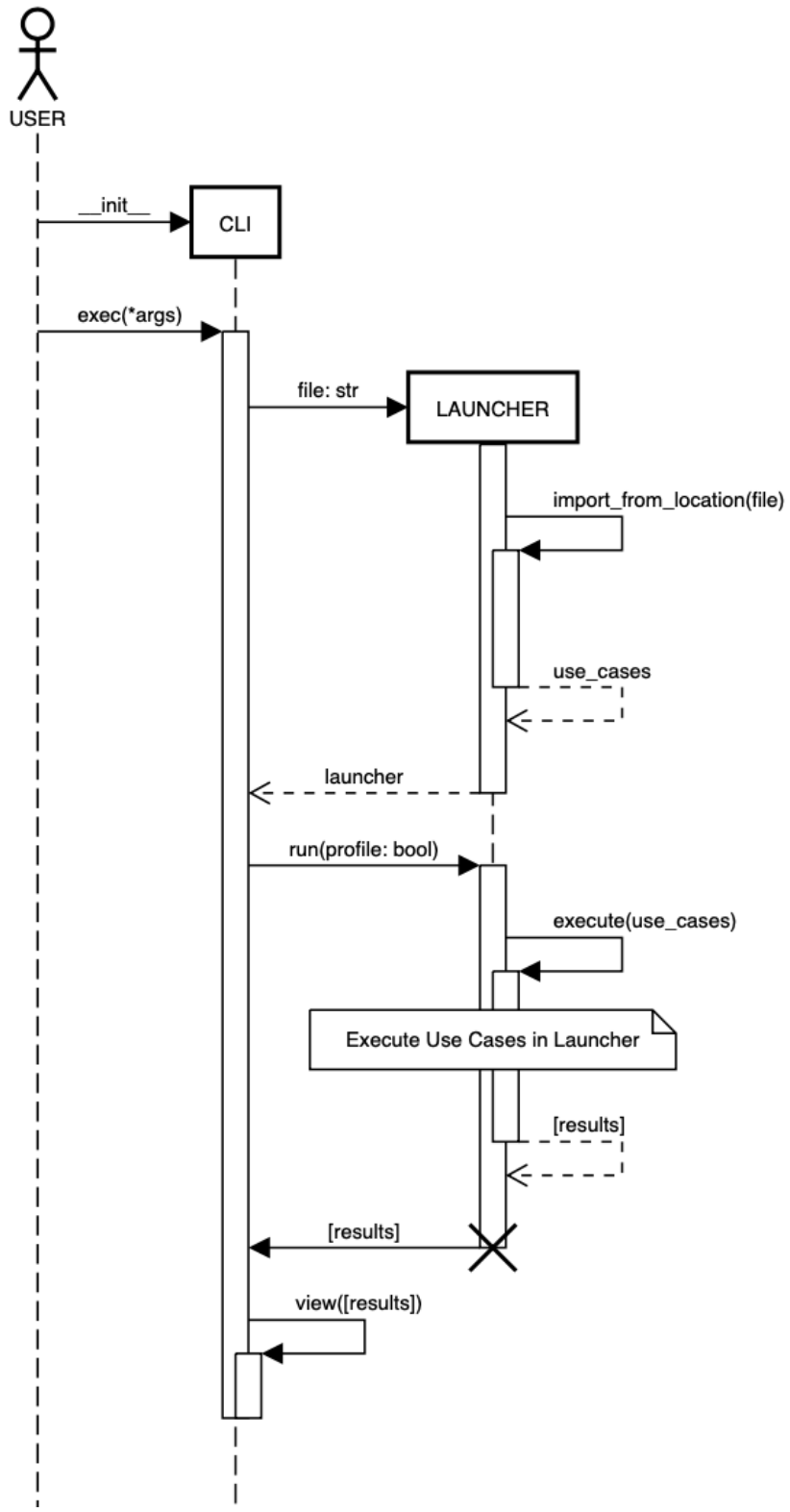


Figure 4.1.: Sequence Diagram: Communication between CLI and Launcher

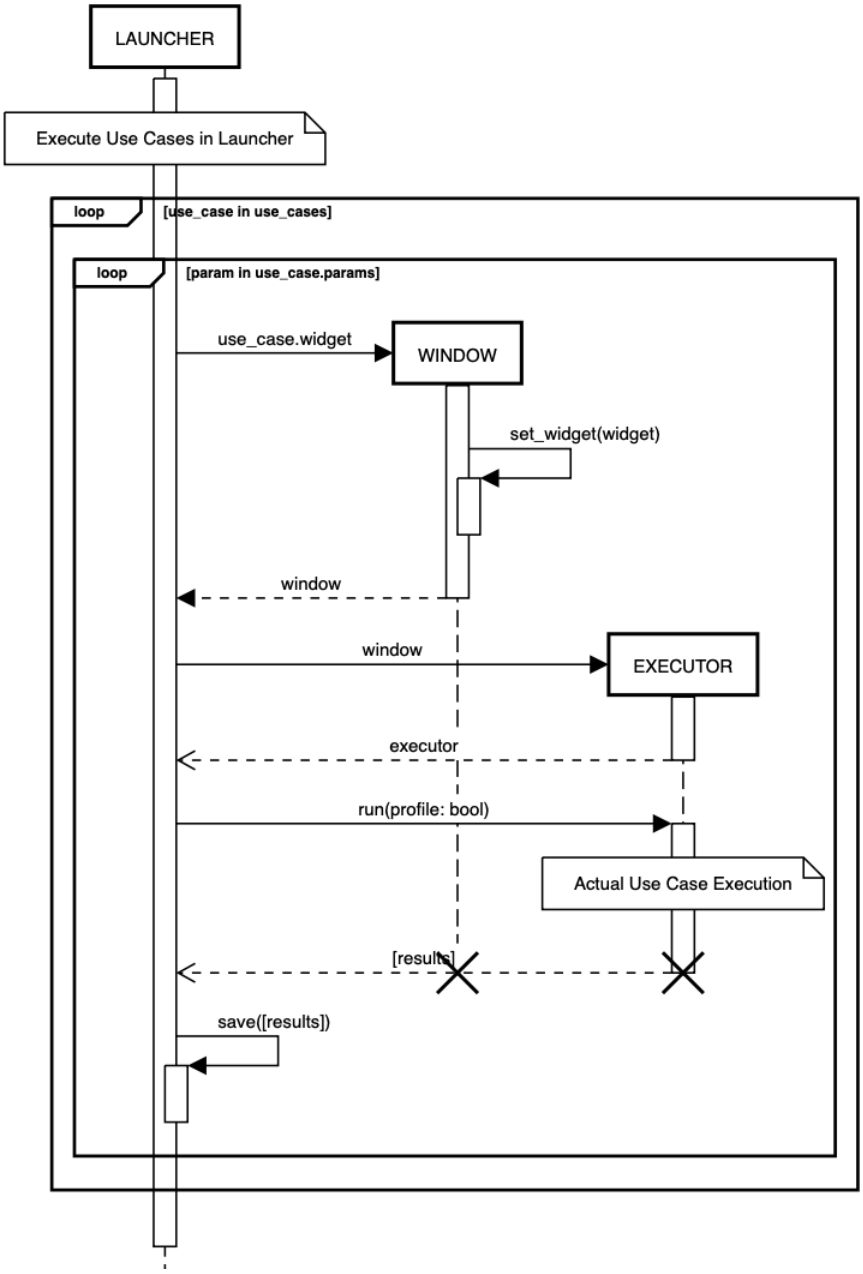


Figure 4.2.: Sequence Diagram: Communication between Launcher, Window and Executor.

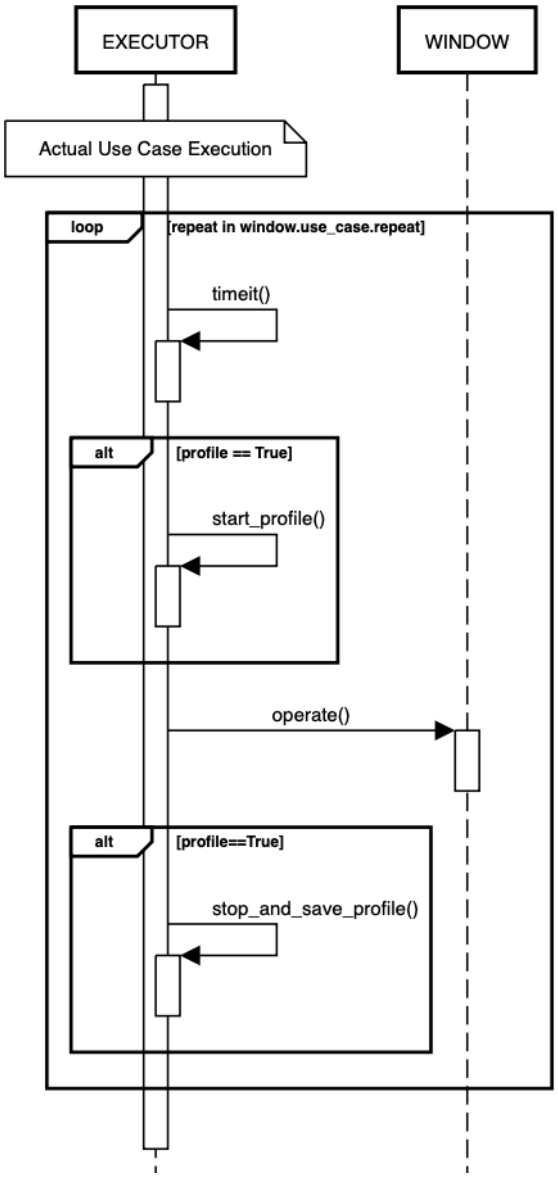


Figure 4.3.: Sequence Diagram: Communication between Executor and Window in more Detail.

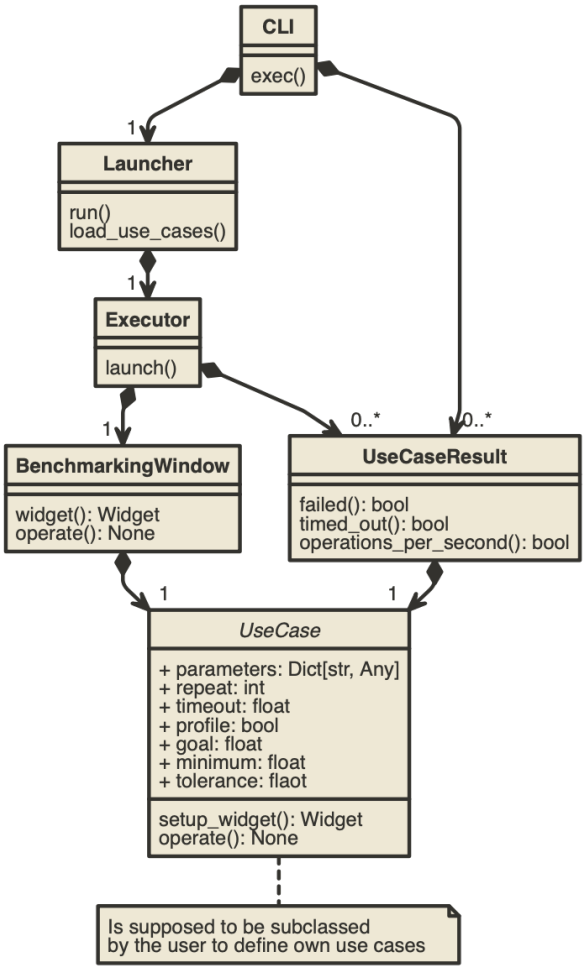


Figure 4.4.: Class Diagram: Classes of the Benchmarking Framework.

4.3.2. Use Case Definition and Plotting Abstraction Layer

In this section, we will have a closer look at the definition of use cases based on the requirements collected in section 4.2. A single use case is represented by a class implementing the `UseCase` interface. One python module can contain a multiple of these use case classes. By only executing classes derived from the use case interface, we can allow defining other classes in the same modules as well without them being executed by accident.

To define plotting benchmarks that are executable with different plotting libraries we will define a unified interface for them. For this work, we will limit this interface to the functionality needed to execute our use cases. In general, this interface is limited to operations supported by all graph libraries implementing it. By using the parameterization functionalities of our use case interface, we can define higher-level plotting benchmarks and execute them using different libraries to compare the results from each of them.

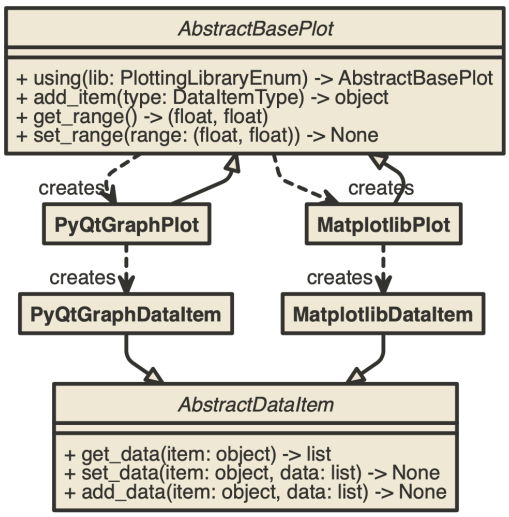


Figure 4.5.: Class Diagram: Classes of the Plotting Abstraction Layer.

Figure 4.5 shows the abstract base classes for the plot widget as well as data items like curves, bar graphs, scatter plots and more. The class `AbstractBasePlot` offers plot functionalities to add a new item, get and set the current view range and a factory method for creating a plot using a specific plotting library. The two subclasses `PyQtGraphPlot` and `MatplotlibPlot` can implement the functionality defined by the base class using their specific Application Programming Interface (API). The `AbstractDataItem` on the other hand defines a common interface for setting, getting and adding new data to a data item of a plot. Similar to the `AbstractBasePlot` , this interface is implemented by the two subclasses `PyQtGraphDataItem` and `MatplotlibDataItem` using their specific API.

Listing 4.3 shows, how the interface should be usable in a standard Qt application. Which library is used for the visualization can be controlled during the widget's initialization using the `AbstractBasePlot` factory method `using()` .

```
1 class Window(QtWidgets.QMainWindow):
2
```

```
    def __init__(self, *args, **kwargs):
        """
        Changing the library passed to the factory method of AbtractBasePlot
        allows switching between the libraries used in for the visualization.
        """
        super().__init__(*args, **kwargs)
        # self.plot = AbstractBasePlot.using(PlottingLibraryEnum.PYQTGRAPH)
        self.plot = AbstractBasePlot.using(PlottingLibraryEnum.MATPLOTLIB)
        # Add Scatter Plot Data Item to the plot.
        item = self.plot.add_item(DataItemType.SCATTER)
        # Set Data of the Scatter Plot
        x = np.linspace(0, 2 * pi, 20)
        y = np.sin(x)
        item.set_data([x, y])
        # Set visible view range
        x_range = [0, 2 * pi]
        y_range = [-1, 1]
        self.plot.set_range([x_range, y_range])
        # Add plot to the window
        self.setCentralWidget(self.plot)
```

Listing 4.3: Creating a plot showing a sinus curve using the Plot Abtraction Layer

4.3.3. User Interface

In this section, we will have a look at the design of the command-line interface based on the requirements we collected in 4.2. Since our command line is supposed to be easily usable for users of other testing frameworks, it should meet the user's expectations. The main purpose of the command line is to start the execution of one or multiple use cases. Listing 4.4 shows how to specify, which use cases should be executed. It should be possible to define a folder, a python module or a specific use case inside a python module. For python modules containing use cases, we will introduce a specific naming convention. The default one should be, that the python module's name is prefixed with `bench_`, similar to test files being often prefixed with `test_`. This pattern should be configurable from the command line as well.

```
# Print Project structure
foo@mbp ~ $: ll my_benchmarks
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot.py
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_image.py
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 b_video.py
# Search files recursively from current working directory
foo@mbp ~ $: widgetmark
    Executing bench_plot.py
    - SinusCurve
    - ZoomingPlot
    Executing bench_image.py
    - HdImage
# Search files recursively from current working directory
```

```
foo@mbp ~ $: widgetmark .
    Executing bench_plot.py
    - SinusCurve
    - ZoomingPlot
    Executing bench_image.py
    - HdImage
# Search files recursively from given folder
foo@mbp ~ $: widgetmark my_benchmarks
    Executing bench_plot.py
    - SinusCurve
    - ZoomingPlot
    Executing bench_image.py
    - HdImage
# Search use cases in given file
foo@mbp ~ $: widgetmark my_benchmarks/bench_plot.py
    Executing bench_plot.py
    - SinusCurve
    - ZoomingPlot
# Execute only single specific use case
foo@mbp ~ $: widgetmark my_benchmarks/bench_plot.py::SinusCurve
    Executing bench_plot.py
    - SinusCurve
# Search for files with different pattern
foo@mbp ~ $: widgetmark --pattern "b_*"
    Executing b_video.py
    - HdVideo
```

Listing 4.4: Executing use cases from the command line

Additionally, the command line should allow us to configure if we want to run the framework with or without profiling, as well as the location the files are saved in. Listing 4.5 shows these configuration options on the command line as well as the generated files.

```
# Start widgetmark and create a profile
foo@mbp ~ $: widgetmark --profile --profile-output=profiles .
    Executing bench_plot.py
    - SinusCurve
    - ZoomingPlot
    Executing bench_image.py
    - HdImage
    Saving 3 profile files to ./profiles
# Profile outputs
foo@mbp ~ $: ll profiles
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot_SinusCurve.profile
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot_ZoomingPlot.profile
    -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_image_HdImage.profile
```

Listing 4.5: Starting benchmarks and create profiles for them

Additionally, the CLI should provide a visualization option for the profiling statistics. Simply printing the profiling statistics to the command line would not be very helpful, since they can grow very fast in size. Because of this, there are multiple tools available,

which offer a much more user-friendly visualization. For our purposes, we decide for *snakeviz*, a browser-based graphical viewer for profiling files. [20]

To improve the usability of the CLI, a help function should be offered which explains what the framework is and how it is going to be used. Listing 4.6 shows how such a help message could look like. Additionally, it should be possible to run the CLI with debug output when searching for bugs in a use case's definition.

```
# Usage explaination for the command line interface
foo@mbp ~ $: widgetmark --help
    usage: widgetmark [-h] [-o PROFILE_FILES_LOCATION] [-p]
                      [--pattern USE_CASE_FILE_NAME_PATTERN] [--visualize]
                      [--loglevel LOGGING_MODULE_LEVEL]
                      [locations [locations ...]]

    A short explaination what widgetmark is.

    Explaination of the individual arguments:
        ...
```

Listing 4.6: Starting benchmarks and create profiles for them

After the execution is completed, the CLI output should show all important information about the found files, the use cases, the performance requirements, the parameters, and the actual reached frame rate. Listing 4.7 shows, how these informations could be presented.

```
# Output after the benchmark execution
foo@mbp ~ $: widgetmark .
---------------------------- WIDGET-MARK ----------------------------
> bench_image.py
  + HdImage              GOAL=30.0, MIN=20.0: .............      21.2
> bench_plot.py
  + SinusCurve
    - PYQTGRAPH          GOAL=30.0, MIN=20.0: .................. 56.6
    - MATPLOTLIB         GOAL=30.0, MIN=20.0: .                   1.3
      - Timed out after 20 seconds.
---------------------------------------------------------------------
Summary (Executed 3 Use Cases)
GOAL        1 Use Cases
MIN         1 Use Cases
NONE        1 Use Cases
---------------------------------------------------------------------
Exceptions  0 Use Cases
Timed Out   1 Use Cases
---------------------------------------------------------------------
```

Listing 4.7: Output of the CLI with showing the benchmark results

5. Implementation of a Benchmark Framework

This chapter will focus on the actual implementation of the framework based on the analysis and design from chapter 4 and the use cases from chapter 3. As a name for our framework we choose *widgetmark*, a composite of the words *widget* and *benchmark*. The language we use is for the implementation is Python 3, which gives us and the framework's users access to newer language features like type hinting. We will begin with an introduction into our project layout, followed by a more detailed look at each sub-package and module of the project.

5.1. Project Structure

The first part of the implementation is setting up our project, whose structure can be seen in figure 5.1. The project will follow python best practices with a package *widgetmark* for the source code and *tests* for unit tests. Additionally the project contains the two generated files *_version.py* and *versioneer.py* for getting version information from version control tags, *README.md* for an introduction into the project, *setup.cfg* for tooling configuration and *setup.py* for dependency definition and installation instructions for `setuptools`.

5.2. Subpackage widgetmark.base

The first sub-package of our implementation is the package `widgetmark.base`. In this package, we will define all base classes for our benchmarking framework, which are not part of the user interface. To keep the framework extensible and not limited to Qt as its GUI framework, we will not yet use Qt specific APIs in this package. Qt specific implementations will be part of the `widgetmark.qt` package.

5.2.1. benchmark.py

The first class for our implementation is the use case interface. To define an interface in Python, we can create an abstract base class without providing an implementation for our functions.

```
1 from abc import ABC, abstractmethod
2 class UseCase(ABC):
```

We distinguish between three different types of information that can be defined in a use case: mandatory functions like widget setup and operation, mandatory attributes like

```
project
├── widgetmark
│   ├── base
│   │   ├── benchmark.py
│   │   ├── executor.py
│   │   └── launcher.py
│   ├── qt
│   │   ├── benchmark.py
│   │   ├── executor.py
│   │   ├── plot.py
│   │   └── util.py
│   ├── cli
│   │   ├── __main__.py
│   │   ├── cli.py
│   │   ├── cli_view.py
│   │   └── loader.py
│   └── _version.py
├── tests
│   ├── __init__.py
│   └── ...
├── setup.py
├── setup.cfg
├── versioneer.py
└── README.md
```

Figure 5.1.: Widgetmark project structure

performance requirements and optional attributes like a timeout. Mandatory functions can be simply implemented as abstract methods using the `abstractmethod` decorator provided by the `abc` package.

```
1    @abstractmethod
2    def operate(self):
3        pass
```

Non-mandatory attributes can be easily implemented as simple class attributes with default values in the use case base class.

```
1    parameters: Dict[str, List] = {}
```

For implementing mandatory class attributes, `abc` does not offer any standard decorators, but we can use `property` in combination with the `abstractmethod` decorator to create abstract properties for our class. While such an implementation does not depict a mandatory attribute on a class level, the abstract property can be implemented as a class attribute in subclasses.

```
1    @property
2    @abstractmethod
```

```
    def goal(self) -> float:
        pass
```

```
class SetTextUseCase(widgetmark.UseCase):
    goal = 30.0
```

With this implementation, there is no visual difference for the user between defining a mandatory or optional attribute. Both, class attributes and abstract properties can also be accessed the same way after the class is initialized using `instance.attribute_name`.

The next class defined in this module is the `UseCaseResult` class, which is based on a single use case. For parameterized use cases, there will be a result instance for each parameter combination. The result object should comprise all the information we have recorded from the execution. To make sure that use cases are read-only after their initialization, we hold the information in private instance attributes and allow reading access only over properties without defining property setters. Additionally, the class offers convenience properties for later checks. An example is a `failed` property that can be used to check if exceptions were raised during the execution.

```
class UseCaseResult:

    def __init__(self,
                 use_case: Union["UseCase", str],
                 operations_per_second: Optional[float] = None,
                 profiling_stats: Optional[pstats.Stats] = None,
                 timed_out: bool = False,
                 exception: Union[Exception, None] = None):
        self._use_case = use_case
        self._ops_ps = operations_per_second
        self._timed_out = timed_out
        self._exception = exception
        self._profiling_stats = profiling_stats
    @property
    def operations_per_second(self):
        return self._ops_ps
```

The last class defined in the module is the benchmarking window, which incorporates the widget defined in the use case and grants access to the operation we want to benchmark. Wrapping the operation allows us to access it later in the executor without having a direct reference to the use case instance itself. Since our goal is to keep the base implementation free from GUI framework-specific API, we won't subclass any Qt window but only define common operations for windows.

5.2.2. executor.py

The executor module contains the class `AbstractBaseExecutor`, which is responsible for executing the benchmarking operations on the window, record the results and create a fitting instance of the `UseCaseResult` class.

As 4.3 shows, for recording execution times, we need a loop like execution of the same operation over and over again. Simply using a loop can lead to unrealistic results,

since we have to give back the control to the event loop in between operations to not skip on pending work caused by our operation. The practical implementation of this is dependent on the used GUI framework and will not be part of this class. Similar to the benchmarking window, this implementation will be done in a fitting executor subclass in package `widgetmark.qt`. As seen in 4.3 an executor instance is initialized for every parameter combination of every use case. After the initialization, a window can be attached to the executor through `set_window()`. The execution can then be launched using the `launch()` method.

```
    def launch(self, profile: bool = False) -> UseCaseResult:
        """Start the execution of the Benchmark"""
        if self._window is None:
            raise Exception("Before launching the Executor a benchmarking "
                            "window has to be set.")
        self._window.use_case.runtime_context = self._runtime_context
        self._last_time = time.time()
        try:
            self._window.integrate_use_case_widget()
        except Exception as e:
            # Catch exceptions raised when executing the Use Case
            logger.debug(traceback.format_exc())
            return UseCaseResult(
                use_case=self._window.use_case,
                exception=e,
            )
        self._window.prepare_window()
        self._launch()
        return cast(UseCaseResult, self._result)
```

Internally the abstract protected function `_launch()` is called, which gives GUI specific subclasses an opportunity to set up the actual execution loop. This loop can use the protected function `_redraw_operation`, which is responsible for calling the operation on the window, collect timing information and control the profiler.

```
    def _redraw_operation(self):
        """
        Execute one operation which triggers the widget redraw.
        This function should be used in the implementation of
        _launch().
        """
        try:
            self._profile()
            self._window.operate()
        except Exception as e:
            # Catch exceptions raised when executing the Use Case
            self._result = self._complete(exception=e)
            logger.debug(traceback.format_exc())
        self._record_current_time()
        self._check_if_completed()
```

To get the current timestamp we can use the `time()` function of the python package `time`. From this timestamp and the last recorded one, we can calculate the delta timing for the last step. The recorded timestamp will be saved in an instance attribute for the next loop step as the last timestamp.

```
    def _record_current_time(self):
        """Save the current timestamp for later evaluation."""
        now = time.time()
        delta = now - self._last_time
        self._last_time = now
        self._timing_recorder.append(delta)
```

The protected function `_profile` is responsible for controlling the profiler. Before executing the operation the profiler will be activated. After the execution, the current profile will be disabled and the stats are added to the previously saved ones.

```
    def _profile(self) -> None:
        """
        Add the last recorded profile to the profiling stats and restart the
        profiler, if the use case has the profile flag set.
        """
        if self.use_case.profile:
            if self._profile_stats is None:
                self._profile_stats = pstats.Stats()
            if self._current_profiler is not None:
                self._current_profiler.disable()
                self._profile_stats.add(self._current_profiler)
            # TODO: use clear() instead of always creating a new profile
            self._current_profiler = cProfile.Profile()
            self._current_profiler.enable()
```

The last important function is `_check_if_completed()`, which stops execution if either the timeout or the repeat counter of the use case is reached. If one is the case, the protected function `_complete()` is called. Since reaching the end of the execution cycle means stopping the execution loop, this step is GUI framework-specific and will be implemented in the subclasses.

```
    def _check_if_completed(self):
        """
        Check if the Benchmark is completed. If the benchmark is completed,
        call the completion handler, which is implemented by the executor
        subclass.
        """
        if self.completed:
            self._result = self._complete()
        elif self.timed_out:
            logger.debug(f"Use case {type(self.use_case).__name__} "
                         f"timed out after taking more than "
                         f"{self.use_case.timeout} seconds.")
            self._result = self._complete(timed_out=True)
        self._execution_counter += 1
```

5.2.3. launcher.py

The module `launcher` contains the class `Launcher`. The launcher can be initialized using either a list of types, which are derived from the `UseCase` class or with a list of file locations, where these classes are defined. If a list of files is passed, the modules are imported using the `importlib` package. To make sure we import only classes that define use cases, we filter the found classes by their type. If the user passes a specific use case name, the found types are additionally filtered by their names.

```
    @staticmethod
    def _import(file_name: str,
                class_type: Type,
                name_filter: str = "") -> List[Type[UseCase]]:
        module_name: str = os.path.basename(file_name)
        logger.debug(f"Search for benchmark in module {module_name}.")
        path: str = file_name
        spec = importlib.util.spec_from_file_location(module_name, path)
        benchmark_module = importlib.util.module_from_spec(spec)
        if (spec.loader is not None
                and isinstance(spec.loader, importlib.abc.Loader)):
            spec.loader.exec_module(benchmark_module)
        else:
            raise InvalidBenchmarkError("Module {module_name} could not"
                                        "be loaded.")
        use_case_classes = Launcher._get_types_from_module(benchmark_module,
                                                           class_type)
        if name_filter:
            logger.debug(f"Filter the found {len(use_case_classes)} "
                         f"use case classes for name {name_filter}")
            use_case_classes = [cls for cls in use_case_classes
                                if re.match(name_filter, cls.__name__)]
        if not use_case_classes:
            raise InvalidBenchmarkError(f"No Use Case class could be "
                                        f"found in module {module_name}.")
        return cast(List[Type[UseCase]], use_case_classes)
```

The initialized launcher instance can be started using the `run()` method, which takes a parameter `profile` for controlling if the executor will create a profile of the use case or not. Before initializing window and executor we create a list of all parameter combinations. For each of these combinations, we initialize the use case class. To grant access in the use case class to these parameters, we set them as instance attributes to the use case instance. If the use case class defines ,for example, the parameter `'number': [0, 1]`, we can access the current value of this parameter in the use case during execution by calling `self.number`.

```
    def run(self, profile: bool = False) -> List[UseCaseResult]:
        """Run all loaded use cases."""
        results: List[UseCaseResult] = []
        for use_case_class in self._use_case_classes:
            combs = self.get_params_combination(use_case_class.parameters)
```

```
            for comb in combs:
                try:
                    use_case: UseCase = use_case_class()
                    if profile:
                        use_case.profile = True
                    params_list = []
                    for name, value in comb.items():
                        setattr(use_case, name, value)
                        params_list.append(Launcher._readable_string(value))
                    use_case.params = ", ".join(params_list)
                except TypeError:
                    results.append(
                        UseCaseResult(
                            use_case=use_case_class.__name__,
                            exception=InvalidUseCaseDefinitionError(),
                        ),
                    )
                    logger.debug(traceback.format_exc())
                    continue
                executor = BackendResolver.executor_type(use_case.backend)()
                window = BackendResolver.window_type(
                    use_case.backend)(use_case=use_case)
                executor.set_window(window=window)
                results.append(executor.launch(profile=profile))
        return results
```

To resolve the right window and executor implementation for the backend defined by the use case, we have the class `BackendResolver` which iterates through a classes subclasses and compares their backend to the backend requested in the use case.

5.3. Subpackage widgetmark.qt

This package contains qt specific implementations of the benchmarking window and the executor, as well as the plotting abstraction layer.

5.3.1. benchmark.py

This module provides a Qt-based implementation of the `AbstractBenchmarkingWindow` based on `QtWidgets.QMainWindow`. A function worth noting here is `make_qt_abc_meta()`. It is responsible for resolving metaclass conflicts between Qt classes and abstract classes by returning a common metaclass for both.

```
class QtWindow(AbstractBenchmarkWindow,
               QMainWindow,
               metaclass=make_qt_abc_meta(QMainWindow)):  # type: ignore
```

5.3.2. executor.py

This module provides a Qt-based implementation of the `AbstractBenchmarkExecutor` based on `QtCore.QObject`. For the execution loop, we will make use of the `QtCore.QTimer` class with a timeout of zero seconds, which will timeout every time it gets the opportunity to. This way we can make sure that control is given back to the event loop each time the use case operation was executed. [38]

To the timer's timeout signal we can connect the executor's base's `_redraw_operation` function, which handles profiling, timing, and operation execution. After the timer is set up we can start the event loop of our Qt application.

```
    def _launch(self):
        self._timer: QTimer = QTimer()
        self._timer.timeout.connect(self._redraw_operation)
        self._timer.start(0)
        self._app.exec()
```

After reaching the repeat counter or the timeout, we can use the `stop()` function of the `QtCore.QTimer` to stop the execution loop.

```
    def _complete(self,
                  timed_out: bool = False,
                  exception: Optional[Exception] = None) -> UseCaseResult:
        self._timer.stop()
        self._app.closeAllWindows()
        self._app.quit()
        self._app.processEvents()
        return UseCaseResult(use_case=self.use_case,
                             operations_per_second=self.operations_per_second,
                             profiling_stats=self._profile_stats,
                             timed_out=timed_out,
                             exception=exception)
```

The Executor is also responsible to start and quit the `QtWidgets.QApplication`. One caveat worth mentioning in this context is, that there should only be a single instance of the application running, since multiple `QApplication` instances will lead to problems. Because of this, the executor class has to make sure, that there is no existing one before creating a new one.

```
    @staticmethod
    def _prepare_qapp():
        """Prepare QApplication instance and install event filters."""
        app = QApplication.instance()
        if app is None:
            app = QApplication([])
            app.installEventFilter(
                InteractionEventFilter.get_event_filter(app))
        return app
```

Since the window and the widget is exposed during the benchmark execution, the user can interact with it. To not create additional work next to the use case, we have to filter

out all use case unrelated operations. As described in section 2.3.3, we can use event filters to filter out unwanted mouse interactions.

```
class InteractionEventFilter(QObject):

    filter = [QEvent.MouseButtonDblClick,
              QEvent.MouseButtonPress,
              QEvent.MouseButtonRelease,
              QEvent.MouseTrackingChange,
              QEvent.GrabMouse,
              QEvent.UngrabMouse,
              QEvent.Move,
              QEvent.DragEnter,
              QEvent.DragLeave,
              QEvent.DragMove]

    def eventFilter(self, o: QObject, e: QEvent):
        if e.type() in InteractionEventFilter.filter:
            e.ignore()
            return True
        return super().eventFilter(o, e)
```

5.3.3. plot.py

The module `widgetmark.qt.plot` contains the implementation of the abstraction layer for plotting operations. This abstraction layer should allow us to perform common actions on different plotting libraries without relying on their library-specific API. The abstraction layer itself is implemented as an abstract class, that provides a factory method that returns the fitting subclass to the passed plotting library.

```
class AbstractBasePlot(
    QtWidgets.QWidget,
    metaclass=make_qt_abc_meta(QtWidgets.QWidget),  # type: ignore
):

    library: PlottingLibraryEnum = None  # type: ignore
    item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {}

    @classmethod
    def using(cls: Type["AbstractBasePlot"],
              library: PlottingLibraryEnum) -> "AbstractBasePlot":
        return cls.get_subclass_for_lib(library=library)()

    @classmethod
    def get_subclass_for_lib(
            cls: Type["AbstractBasePlot"],
            library: Union[PlottingLibraryEnum, str],
    ) -> Type["AbstractBasePlot"]:
        fitting_classes: List[Type[AbstractBasePlot]] = [
            c for c in AbstractBasePlot.all_subclasses(cls)
```

```
            if library == c.library and c.library is not None
        ]
        if not fitting_classes:
            raise ValueError(f"No fitting subclass was found for the "
                             f"plot library {library}.")
        elif len(fitting_classes) > 1:
            logger.debug(f"Multiple subclasses were found for the "
                         f"plot library {library}.")
        return fitting_classes[0]
```

Each operation we want to use will get its own method in this abstract class. As an example we will have a look at the `add_item` function, which allows us to add a new item to the plot, which can display some data.

```
    @abstractmethod
    def add_item(self, item_type: DataItemType) -> "AbstractDataItem":
        pass
```

For PyQtGraph we can implement the `add_item` function based on the `PlotDataItem` .

```
class PyQtGraphPlot(pg.PlotWidget, AbstractBasePlot):

    library = PlottingLibraryEnum.PYQTGRAPH

    item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {
        DataItemType.CURVE: (pg.PlotDataItem, {"pen": "w", "symbol": None}),
        DataItemType.SCATTER: (pg.PlotDataItem, {"pen": None, "symbol": "o"}),
    }
    def add_item(self, item_type) -> "PyQtGraphDataItem":
        item_class, params = self.item_type_mapping.get(item_type, (None, {}))
        if item_class is not None and isinstance(params, Dict):
            item = item_class(**params)
            self.addItem(item)
            return PyQtGraphDataItem(item)
        else:
            raise ValueError(f"For data item type {item_type} is no fitting "
                             f"item is defined.")
```

For Matplotlib, on the other hand, we can use the `Line2D` object for implementing the same function.

```
class MatPlotLibPlot(AbstractBasePlot):

    library = PlottingLibraryEnum.MATPLOTLIB

    item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {
        DataItemType.CURVE: (Line2D, {
            "xdata": [],
            "ydata": [],
            "linestyle": "-",
        }),
```

```
        DataItemType.SCATTER: (Line2D, {
            "xdata": [],
            "ydata": [],
            "linestyle": "",
            "marker": "o",
        }),
    }

    def add_item(self, item_type) -> "MatPlotLibDataItem":
        item_class, params = self.item_type_mapping.get(item_type, (None, {}))
        if item_class is not None and isinstance(params, Dict):
            item = item_class(**params)
            self._plot.add_artist(item)
            return MatPlotLibDataItem(self._canvas, item)
        else:
            raise ValueError(f"For data item type {item_type} is no fitting "
                             f"item is defined.")
```

Both functions will return objects derived from the same `AbstractDataItem`, which allows reading, writing and adding data in the form of a two-dimensional NumPy array containing x and y-values. In our use cases, we can add the plotting library we want to benchmark as a parameter and pass it to the factory function `using()` of the `AbstractBasePlot` class. This way we can benchmark the same use case in two completely different plotting libraries without having to define two separate use cases.

5.4. Subpackage widgetmark.cli

The last package of our project's source code contains all modules related to the CLI. Its purpose is accepting user input to start the launcher as well as presenting the results. Additionally, it saves the recorded profiling statistics to separate files and starts the visualization using `snakeviz`. For defining command line arguments, the python package `argsparse` is used, which will automatically add a help option, which explains the CLI's usage when the user executes `widgetmark --help`.

When installing `widgetmark` using `setuptools`, we can register our CLI as a console script. This allows us to make a Python function accessible from the command line with a specific name. In the module `main.py`, we have our central main method, which initializes the CLI and executes it. We will register this one with the package's name *widgetmark*.

```
def main():
    """Initialize the CLI and start it."""
    CLI().exec()
```

```
setup(
    name="widgetmark",
    entry_points={
        "console_scripts": ["widgetmark = widgetmark.cli:main"],
    },
```

```
)
```

6. Evaluation

In this chapter, we will evaluate the framework we designed and developed in the last two chapters. We will start by evaluating, if the implementation fits our requirements, by implementing the use cases presented in chapter 3 and investigating their results. This will be followed by an evaluation of the performance overhead the framework is producing, by comparing the results with a minimal PyQt application recreating the same use case.

6.1. Use Case Implementation

This section will focus on the implementation of the use cases described in chapter 3 based on widgetmark and its plotting library abstraction layer. To have an overview of the development of the performance of each use case depending on the increasing plot counts, curve counts or dataset sizes, we will run each use case with different parameters. We will start with smaller sizes and increase them until we reach the use cases demanded requirements. As a minimum refresh rate, we will take the update frequency of the use case. We will set the goal frame rate to 60Hz, which matches most modern monitors. As data we will take random points each redraw, so we can be sure that we will get a full redraw every time. As a repeat count, we will take a reasonable big number of 2000 redraws. Additionally, we will define a timeout that can stop use cases that take an unexpectedly long amount of time. Each use case will be tested against both PyQtGraph and Matplotlib.

All of our three use cases have a very common setup. To avoid duplicated code, we will define a setup helper class next to the use cases, which each one can take advantage of. The implementation of its widget setup function can be seen in listing 6.1. This helper function is responsible for initializing a given number of plots, curves, and random data sets.

```
    @staticmethod
    def quick_setup(lib: widgetmark.PlottingLibraryEnum,
                    item_type: widgetmark.DataItemType,
                    item_amount: List[int],
                    ds_length: int,
                    ds_count: int) -> Tuple[List[widgetmark.AbstractBasePlot],
                                            List[widgetmark.AbstractDataItem],
                                            List[np.ndarray]]:
        plots = []
        curves = []
        data = []
        for pc in item_amount:
            plot = widgetmark.AbstractBasePlot.using(lib)
            plot.set_range(((0, 10), (0, pc)))
```

```
            plots.append(plot)
            for i, _ic in enumerate(range(pc)):
                bounds = [0, 10, i, i + 1]
                d = UseCaseHelper._prepare_data(ds_count,
                                                ds_length,
                                                *bounds)
                data.append(d)
                curves.append(plot.add_item(item_type=item_type))
        return plots, curves, data
```

Listing 6.1: Setup helper function for creating plots

6.1.1. Distributed Oscilloscope for BE-CO-HT

The Distributed Oscilloscope features a central plot containing up to 8 curves displaying up to 100,000 points at a time. To handle the update frequency of 25 Hertz, we will need a minimum frame rate of 25 frames per second. Listing 6.4 shows the implementation of these parameters in a use case class as class attributes.

```
class HtUseCase(UseCaseHelper, widgetmark.UseCase):

    backend = widgetmark.GuiBackend.QT
    goal = 50.0
    minimum = 25.0
    tolerance = 0.05
    repeat = 1000
    timeout = 100
    # These will be available during runtime as instance attributes
    parameters = {"plot_lib": list(widgetmark.PlottingLibraryEnum),
                  "data_count": [1000, 10000, 100000],
                  "curve_count": [1, 4, 8]}
```

Listing 6.2: Definition of the parameters for the BE-CO-HT use case.

Listing 6.3 shows the implementation of the widget setup and the operation. For the setup we use the helper function defined in listing 6.1. Since we only create a single plot, we return it as the widget to use. The next function is our operation definition, where we choose data and display in our created plots. To choose data from our random data sets, we take the current redraw run as an index, which can be accessed through the runtime context object, that every use case has access to.

```
    def setup_widget(self):
        """Setup widgets for the """
        self._ds_count = 5
        self._plots, self._curves, self._data = self.quick_setup(
            lib=self.plot_lib,
            item_type=widgetmark.DataItemType.CURVE,
            item_amount=[self.curve_count],
            ds_length=self.data_count,
            ds_count=self._ds_count,
        )
```

```
        return self._plots[0]

    def operate(self):
        for ci, c in enumerate(self._curves):
            di = self.runtime_context.current_run % self._ds_count
            c.set_data(self._data[ci][di])
```

Listing 6.3: Widget setup and operation definition for the BE-CO-HT use case.

6.1.2. Monitoring Application for BE-OP-LHC

Our second use case features one central plot containing up to 3000 curves displaying up to 2 * 3600 points, which should be updated every second. Listing 6.4 shows the implementation of these requirements as parameters in the use case class. Widget Setup and operation definition do not differ from listing 6.3.

```
class LhcUseCase(UseCaseHelper, widgetmark.UseCase):

    backend = widgetmark.GuiBackend.QT
    goal = 50.0
    minimum = 1.0
    tolerance = 0.05
    repeat = 1000
    timeout = 100
    # These will be available during runtime as instance attributes
    parameters = {"plot_lib": list(widgetmark.PlottingLibraryEnum),
                  "data_count": [360, 720, 3600, 7200],
                  "curve_count": [300, 3000]}
```

Listing 6.4: Definition of the parameters for the BE-OP-LHC use case.

6.1.3. Linac4 Source GUI for BE-CO-APS

Compared to the prior two use cases, this use case features multiple plots, which means we can extend our parameter list with another entry for the plot count. The implementation of the parameters in the use case class is displayed in 6.5.

```
class ApsUseCase(UseCaseHelper, widgetmark.UseCase):

    backend = widgetmark.GuiBackend.QT
    goal = 50.0
    minimum = 30.0
    tolerance = 0.05
    repeat = 1000
    timeout = 100
    # These will be available during runtime as instance attributes
    parameters = {"plot_lib": list(widgetmark.PlottingLibraryEnum),
                  "data_count": [3600 / 1.2],
                  "curve_count": [3],
```

```
                "plot_count": [(1, 1), (4, 2)]}
```

Listing 6.5: Definition of the parameters for the BE-CO-APS use case.

Using multiple Plots means as well, that we have to change our setup function slightly as seen in 6.6 by wrapping the plots in another `QtWidgets.QWidget` object, which has a `QtWidgets.QGridLayout` set as its internal layout. This way we can pass our plots bundled as one single widget to the benchmark window.

```
    def setup_widget(self):
        """Wrap multiple plots inside a QWidget"""
        self._ds_count = 5
        self._plots, self._curves, self._data = self.quick_setup(
            lib=self.plot_lib,
            item_type=widgetmark.DataItemType.SCATTER,
            item_amount=[self.curve_count for _ in range(self.plot_count[0])],
            ds_length=self.data_count,
            ds_count=self._ds_count,
        )
        widget = QWidget()
        layout = QGridLayout()
        widget.setLayout(layout)
        for index, plot in enumerate(self._plots):
            row = int(index / self.plot_count[1])
            column = index % self.plot_count[1]
            layout.addWidget(plot, row, column)
        return widget
```

Listing 6.6: Widget setup and operation definition for the BE-CO-APS use case.

6.1.4. Results

After our benchmarks are defined, we can execute them using the widgetmark command-line interface. To receive results closest to the conditions in the later production environment, we will run the benchmarks on machines with similar hardware configurations.

CPU: Intel Core i7 6700 with 4 Cores (8 Threads), 3.4GHz Base Clockspeed, 4.0GHz Maximum Clockspeed

RAM: 32GB

GPU: Intel HD 530 Integrated Graphics

OS: 64bit CentOS7

The size of the testing window was 800 x 600 pixels on all executed use cases. The tables 6.1, 6.2 and 6.3 show the recorded results on this machine. As we can see, PyQtGraph does perform substantially better compared to Matplotlib, but the use cases reveal that both have their limits. Especially on the load provided by the BE-OP-LHC use case, both are struggling severely with frame rates as low as 0.0 FPS, which means that a single redraw

of the entire dataset was taking more than 20 seconds on average. Interaction with the plot is impossible at this point. For both libraries, the number of points or datasets should be heavily reduced in these scenarios to keep the user interface interactive. From the two given libraries, PyQtGraph is still the better choice when it comes to representing large data sets, if interactive rendering speeds are required.

Table 6.1.: Results of the BE-CO-HT use case

Use Case	Library	Plots	Items	Type	Dataset	Frame Rate
BE-CO-HT	PyQtGraph	1	1	Curve	1000	861.6
BE-CO-HT	Matplotlib	1	1	Curve	1000	31.0
BE-CO-HT	PyQtGraph	1	1	Curve	10000	150.7
BE-CO-HT	Matplotlib	1	1	Curve	10000	5.2
BE-CO-HT	PyQtGraph	1	1	Curve	100000	24.7
BE-CO-HT	Matplotlib	1	1	Curve	100000	1.6
BE-CO-HT	PyQtGraph	1	4	Curve	1000	434.9
BE-CO-HT	Matplotlib	1	4	Curve	1000	7.7
BE-CO-HT	PyQtGraph	1	4	Curve	10000	95.6
BE-CO-HT	Matplotlib	1	4	Curve	10000	1.6
BE-CO-HT	PyQtGraph	1	4	Curve	100000	9.3
BE-CO-HT	Matplotlib	1	4	Curve	100000	0.5
BE-CO-HT	PyQtGraph	1	8	Curve	1000	256.5
BE-CO-HT	Matplotlib	1	8	Curve	1000	3.6
BE-CO-HT	PyQtGraph	1	8	Curve	10000	59.2
BE-CO-HT	Matplotlib	1	8	Curve	10000	0.8
BE-CO-HT	PyQtGraph	1	8	Curve	100000	5.8
BE-CO-HT	Matplotlib	1	8	Curve	100000	0.2

Table 6.2.: Results of the BE-OP-LHC use case

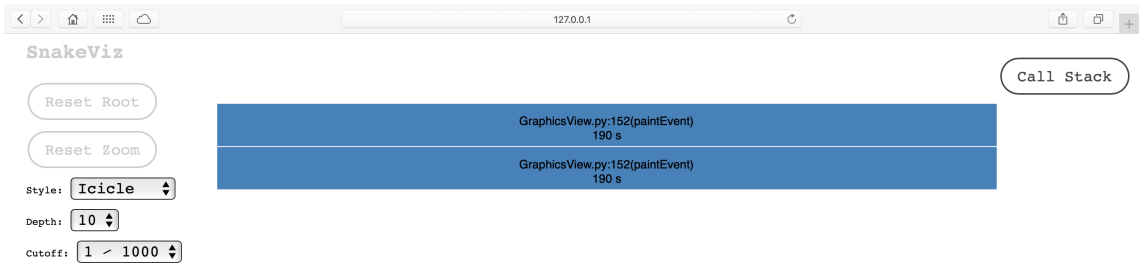
Use Case	Library	Plots	Items	Type	Dataset	Frame Rate
BE-OP-LHC	PyQtGraph	1	300	Curve	720	10.5
BE-OP-LHC	Matplotlib	1	300	Curve	720	0.0
BE-OP-LHC	PyQtGraph	1	3000	Curve	720	2.6
BE-OP-LHC	Matplotlib	1	3000	Curve	720	0.0
BE-OP-LHC	PyQtGraph	1	3000	Curve	7200	0.9
BE-OP-LHC	Matplotlib	1	3000	Curve	7200	0.0

With the profiler activated we can investigate where most of the computing time is spent. In the following we will, for example, have a look at the profiles created for the use cases run with PyQtGraph with high amounts of data. For plots containing curves with a high amount of data, this is mainly the paint event handler of the graphics view, which handles repainting the plot every time the plot gets changed or moved. In more detail, the

Table 6.3.: Results of the BE-CO-APS use case

Use Case	Library	Plots	Items	Type	Dataset	Frame Rate
BE-CO-APS	PyQtGraph	1	3	Scatter	3000	15.9
BE-CO-APS	Matplotlib	1	3	Scatter	3000	9.6
BE-CO-APS	PyQtGraph	4	3	Scatter	3000	3.9
BE-CO-APS	Matplotlib	4	3	Scatter	3000	2.4

translation of the raw data to a `QtGui.QPainterPath` as well as the actual painting done by Qt's GraphicsView Framework. Screenshot 6.1 shows the profile of the BE-CO-HT use case presented by Snakeviz in the Browser.



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
8888	103.1	0.0116	103.3	0.01162	functions.py:1324(arrayToQPath)
9280	2.829	0.0003048	3.64	0.0003922	PlotCurveItem.py:309(updateData)
36376	0.2908	7.995e-06	0.4943	1.359e-05	functions.py:249(mkPen)
8896	0.172	1.934e-05	188.9	0.02123	PlotCurveItem.py:418(paint)

Figure 6.1.: Screenshot of the BE-CO-HT use case's profile visualized in snakeviz

When it comes to scatter plots, another function does take quite some time as well. A big amount of time is spent in the preparation of the symbol atlas, which is a prerendered image of all symbols used in the scatter plot. When it comes to rendering the actual plot, the already rendered symbols can simply be copied from this atlas and pasted into the image. One potential way of improving performance here is to bypass the loop over each data point in case all symbols are the same. Screenshot 6.2 shows the profile of the BE-CO-APS use case presented by Snakeviz in the browser.

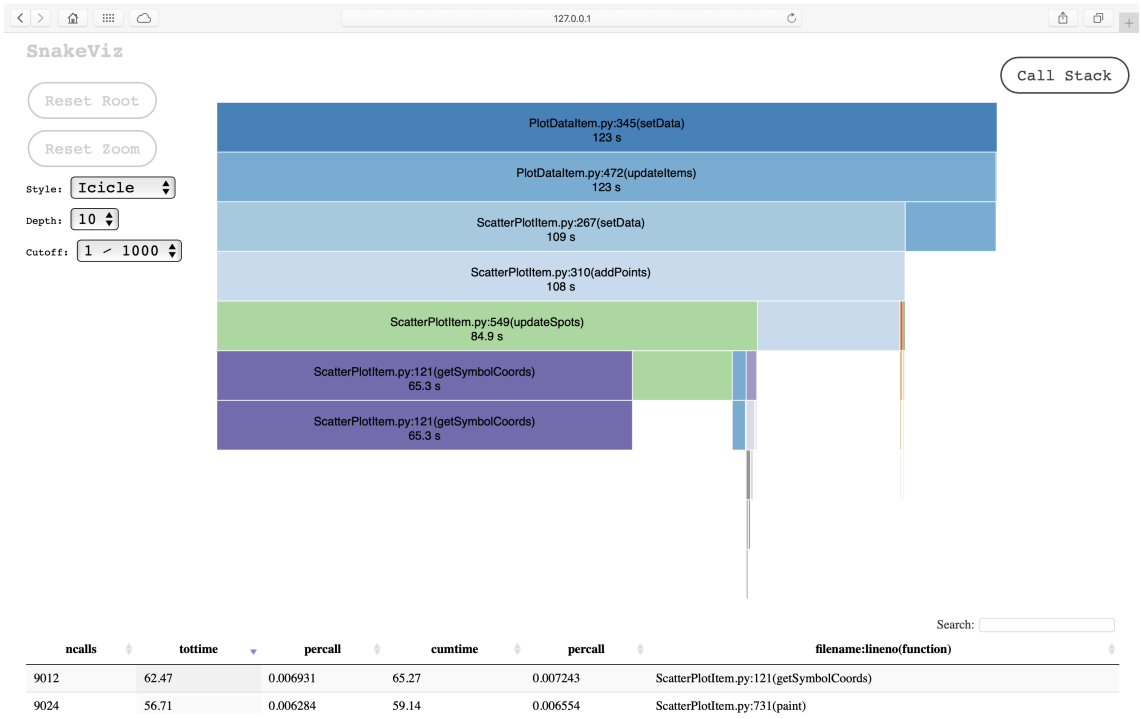


Figure 6.2.: Screenshot of the BE-CO-APS use case's profile visualized in snakeviz

6.2. Performance Overhead

This chapter will focus on the evaluation of the framework. The biggest question when it comes to evaluating the performance of widgetmark is, how much overhead it is adding compared to a standard application. The bigger our overhead is, the more negative the results will be compared to the real value we can expect.

6.2.1. Comparison Application

To evaluate the accuracy of our benchmarking framework, we will compare the times measured in a simple example use case with an actual implementation as a minimal PyQt application. This application will only house a single plot updated by the same conditions as the use case. To measure the timing in the application, we will record a timestamp after each update. To find out how the overhead of the framework develops under different load scenarios, we will use different dataset size parameters for the plotting. For the evaluation, we will only choose a single plotting library, which will be PyQtGraph. The source code for this application can be seen in listing A.3. It is comprised of a single main window, containing the plot as its central widget. The plot is a `pyqtgraph.PlotWidget()`, which is updated using a `QtCore.QTimer` with a zero-second timeout, similar to widgetmark's Qt executor class. Next to the data update, the current timestamp is recorded on every timer timeout. As a hardcoded barrier, 1000 redraws are chosen. For every configuration, we have to execute the python module with different parameters. The source code for the use case is appended in listing A.4. In both cases, the data generation is excluded from time recording.

6.2.2. Delta Time Accuracy

Table 6.4 compares the measured frame rates for each use case. Both variants were executed on the same hardware with the same screen resolution as well as the same window size. The measured data shows a trend of the framework being slightly slower compared to the minimal PyQt application.

Table 6.4.: Frame rate comparison between widgetmark and the comparison application.

Dataset Size	Widgetmark	PyQt Application	Overhead
1000	58.8	59.8	2.2%
10000	59.7	59.6	0%
500000	23.8	25.9	8.1%
1000000	12.5	13.9	10.1%

While these measurements already provide us with a general trend of the framework adding a small overhead to our execution, it raises the question, how this deviation compares with the fluctuation of the recorded times within an execution. To compare this we first will have to get access to all measured delta times for the PyQt application as well as the widgetmark run.

As a small rehearsal, we will quickly explain the sizes used in the following detailed evaluation and why they are meaningful for us. The first important measurement is the delta timing Δt, which describes the difference between two measured adjacent time stamps. From a set of m timestamps it can be calculated for two adjacent timestamps t_{i+1} and t_i as:

$$\Delta t_i = t_{i+1} - t_i$$

The arithmetic mean or average $\bar{\Delta t}$ describes the average from our set of $m-1$ measured delta times and is defined as:

$$\bar{\Delta t} = \frac{1}{m-1} * \sum_{i=1}^{m-1} \Delta t_i$$

The standard deviation σ of our delta times describes, how far values are deviated on average from $\bar{\Delta t}$. From the standard σ we can get a better idea, if the measured times are very consistent or if they differ much from each other. It is defined as:

$$\sigma_{\Delta t} = \sqrt{\frac{1}{m-2} * \sum_{i=1}^{m-1} (\Delta t_i - \bar{\Delta t})^2}$$

The range of a set of $m-1$ delta times Δt_i describes the difference between the largest and smallest measured Δt and is defined as:

$$R_{\Delta t} = \Delta t_{max} - \Delta t_{min}$$

Figures A.5, A.6, A.7 and A.8 compare all measured delta times in the widgetmark as well as the minimal PyQt application. For smaller dataset sizes, there are only small difference between both noticeable. If the dataset size however increases, the difference measured in the frame rates can be seen very well. In figure 6.3, which shows the average of the measured delta times, this difference becomes more apparent.

Figure 6.4 compares the difference between the average delta times, the fluctuation of measured individual delta times and the ranges $R_{\Delta t}$ of the framework as well as the minimal application. The difference between widgetmark and the app in $\bar{\Delta t}$ is bigger compared to their individual standard deviations $\sigma_{\Delta t}$. If it was in this deviation range, it would be much more likely to be just a fluctuation in our measurements. By clearly exceeding the expected deviation, we can assume that the overhead will be reproducible, which supports our assumption, that widgetmark does add additional overhead to the execution, especially for larger data set sizes. Compared to the measured value ranges $R_{\Delta t}$ of each run, the difference is considerably smaller. These large ranges clearly show, that it is very hard to further estimate, how high this actual overhead will be on a run since the recorded delta times can be very heavily influenced by factors like the current load on the system. This is especially well represented by the exceptional high range for the comparison application at a data set size of 50000 points. Additionally, it shows the importance of a reasonably high repeat counter, to keep the influence of such stray bullets as small as possible.

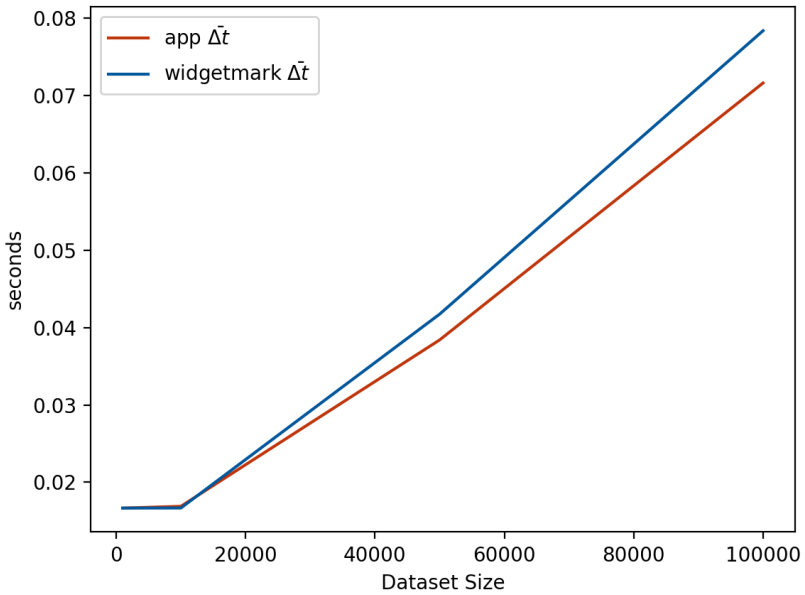


Figure 6.3.: Average Delta Timing of Widgetmark and the Comparison Application

As a summary of the evaluation we can see, that widgetmark does add a slight overhead to the executed use case. This overhead, however, does only slightly influence the outcome of each use case, since it still manages to represent the library's realistic performance capabilities.

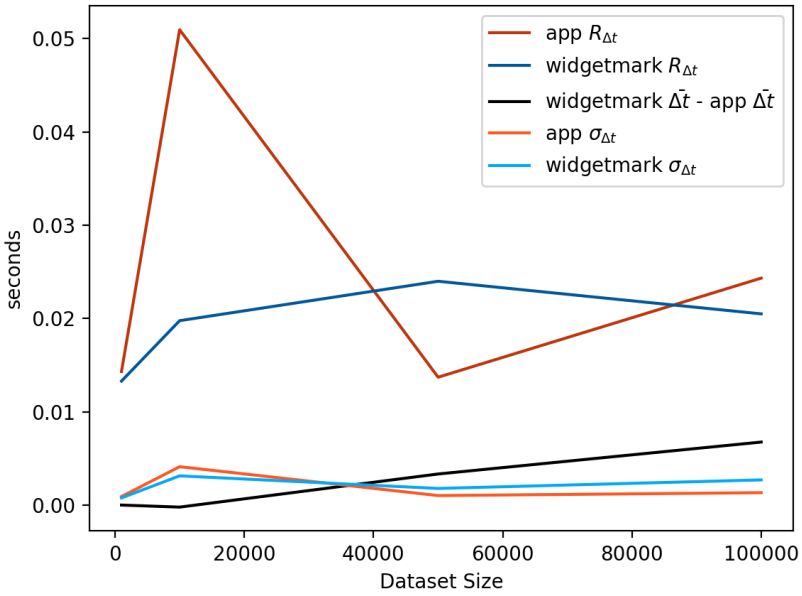


Figure 6.4.: Standard Deviation, Delta Time Value Ranges and average Delta Times of Widgetmark and the Comparison Application

7. Conclusion

This chapter will provide a summary of this work as well as an outlook for further improvements.

7.1. Summary

The purpose of this work was to analyze the performance requirements for python plotting libraries at CERN, developing a design and implement a solution that allows use case driven performance benchmarking.

In the beginning, we had a look at all major technologies and topics involved in the later solution to create a common knowledge base for the following chapters. We started with a general introduction into data visualization, explored different types of benchmarking software to learn about performance testing and discussed the fundamental principles of the Qt application framework with Widgets, Layouts, Signals, Slots as well as the Event System.

Following that, we investigated use cases for plotting libraries at CERN originating from three different applications. Based on the expected update frequencies of each use case we chose the frame rate as our metric of choice to describe the rendering performance. A summary of the profiling options in Python presented a useful tooling option to discover performance bottlenecks in end-user applications.

Using the findings from the previous chapters, we further analyzed the requirements for defining each use case and designed a solution that allowed us to depict them. Our focus for the implementation was providing a use case definition interface that was easy to use, capable enough to cover all collected use cases and easily parametrizable, an user-friendly CLI for executing multiple use cases, as well as a helpful representation of the recorded results.

To investigate, how close the results reached the later production application the use case originated from, we compared the results of different use cases with a minimal example application. Even though we could measure an overhead produced by the framework during the execution, the results could still realistically represent performance we could expect.

7.2. Outlook

Based on the developed *widgetmark* framework, users can conveniently create own use cases for their own plotting needs and easily measure the performance it reached in different plotting libraries. This allows them to choose a fitting plotting library not only based on the offered features but also on the performance capabilities they need.

While its use at this point is relatively narrow, there are multiple opportunities to extend the purpose of the framework. For other GUI Frameworks, new backends could be implemented, for more plotting libraries the plotting abstraction layer can be extended and for other use case scenarios, the use case interface could be extended for example with a setup and tear down phase. To keep widgetmark as modular as possible, it could be extended by plug-in functionalities, which would allow extending the framework more easily without having to change the base implementation.

Another area of improvement is minimizing the overhead added by the framework during the execution. By minimizing this overhead the results could be even closer to the later real-life performances.

Additionally, the findings in analysis and design can bring insights into other areas of software performance evaluation and can be a template for other benchmarking frameworks that do not focus on GUI widgets.

Bibliography

[1] Ajitsaria Abhinav. *What is the Python Global Interpreter Lock (GIL)?* June 2018. URL: https://realpython.com/python-gil/.

[2] Jeff Atwood. *Performance is a Feature*. June 2011. URL: https://blog.codinghorror.com/performance-is-a-feature/.

[3] "Benchmarking als Instrument eines wettbewerbsorientierten Performance Management". In: *Marketing Performance*. Gabler, pp. 237–249. DOI: 10.1007/978-3-8349-0664-9_11. URL: https://doi.org/10.1007/978-3-8349-0664-9_11.

[4] Jasmin Blanchette. *Another Look at Events*. 2004. URL: https://doc.qt.io/archives/qq/qq11-events.html#eventhandlingandfiltering.

[5] Luke Campagnola. *PyQtGraph Documentation*. 2011. URL: http://www.pyqtgraph.org/documentation/index.html.

[6] Drew Campbell. *Understanding Delta Timing*. July 2019. URL: https://medium.com/@dr3wc/understanding-delta-time-b53bf4781a03.

[7] CERN. *A day in the CERN Control Center*. URL: https://home.cern/news/news/accelerators/day-cern-control-centre.

[8] CERN. *A day in the CERN Control Center*. URL: https://www.heise.de/newsticker/meldung/Nvidia-GeForce-GTX-2080-und-2080-Ti-Alle-Herstellerkarten-im-Ueberblick-4142326.html.

[9] CERN. *Beams Department*. URL: https://beams.web.cern.ch.

[10] CERN. *Hardware and Timing section*. 2019. URL: https://espace.cern.ch/be-dep-old/CO/Hardware_Timing/default.aspx.

[11] CERN. *Home*. URL: https://home.cern.

[12] CERN. *Lhc Report: The final days of Run 2*. URL: https://home.cern/news/news/accelerators/lhc-report-final-days-run-2.

[13] CERN. *Linear Accelerator 4*. URL: https://home.cern/science/accelerators/linear-accelerator-4.

[14] CERN. *Our history*. URL: https://home.cern/about/who-we-are/our-history.

[15] CERN. *Processing: What to record?* URL: https://home.cern/science/computing/processing-what-record.

[16] CERN. *The Large Hadron Collider*. URL: https://home.cern/science/accelerators/large-hadron-collider.

[17] CERN. *The Low Energy Ion Ring*. URL: https://home.cern/science/accelerators/low-energy-ion-ring.

[18] CERN. *Who we are*. URL: https://home.cern/about/who-we-are.

[19] Julien Danjou. *Profiling Python using cProfile: a concrete case*. Nov. 2019. URL: https://julien.danjou.info/guide-to-python-profiling-cprofile-concrete-case-carbonara/.

[20] Matt Davis. *Snakeviz*. URL: https://jiffyclub.github.io/snakeviz/.

[21] Stéphane Deghaye and Eve Fortescue-Beck. *Introduction to the BE-CO Control System*. 2019 Edition. CERN, 2019.

[22] Ben Dotson. *How Particle Accelerators Work*. 2014. URL: https://www.energy.gov/articles/how-particle-accelerators-work.

[23] ATLAS Experiment. *Live Collisions in the ATLAS Detector*. URL: https://atlas-public.web.cern.ch/atlas-live.

[24] ATLAS Experiment. *Trigger and Data Acquisition System*. URL: https://atlas-public.web.cern.ch/discover/detector/trigger-daq.

[25] Stephen Few. "Data Visualization - Past, Present, and Future". In: (2007). URL: https://www.perceptualedge.com/articles/Whitepapers/Data_Visualization.pdf.

[26] Michael Firendly. *Milestones in the History of Data Visualization*. Nov. 2003. URL: http://datavis.ca/papers/carme2003-2x2.pdf.

[27] Python Software Foundation. *Extending Python with C or C++*. 2019. URL: https://docs.python.org/3/extending/extending.html.

[28] Python Software Foundation. *The Python Profilers*. 2020. URL: https://docs.python.org/3.8/library/profile.html.

[29] Python Software Foundation. *Thread State and the Global Interpreter Lock*. 2019. URL: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[30] Michael Friendly. "Milestones in the History of Data Visualization: A Case Study in Statistical Historiography". In: *Classification: The Ubiquitous Challange*. Ed. by C. Weihs and W. Gaul. New York: Springer, 2005, pp. 34–52. URL: http://www.math.yorku.ca/SCS/Papers/gfkl.pdf.

[31] Robert B. Haber and David A. McNabb. "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems". In: ().

[32] John D. Hunter. *History*. 2008. URL: https://matplotlib.org/3.1.1/users/history.html.

[33] G. Kruk and M. Peryt. *JDATAVIEWER – JAVA-BASED CHARTING LIBRARY*. CERN, Geneva, Switzerland, Nov. 2009.

[34] Riverbank Computing Limited. *What is PyQt?* 2018. URL: https://www.riverbankcomputing.com/software/pyqt/intro.

[35] Riverbank Computing Limited. *What is SIP?* 2018. URL: https://www.riverbankcomputing.com/software/sip/intro.

[36] The Qt Company Ltd. *Graphics View Framework | Qt Widgets 5.14.0*. 2019. URL: https://doc.qt.io/qt-5/graphicsview.html.

[37] The Qt Company Ltd. *Qt Namespaces | Qt Core 5.14.0; enum Qt::ConnectionType*. 2019. URL: https://doc.qt.io/qt-5/qt.html#ConnectionType-enum.

[38] The Qt Company Ltd. *QTimer class| Qt Widgets 5.14.0*. 2019. URL: https://doc.qt.io/qt-5/qtimer.html#details.

[39] The Qt Company Ltd. *Signals and Slots | Qt Core 5.14.0*. 2019. URL: https://doc.qt.io/qt-5/signalsandslots.html.

[40] The Qt Company Ltd. *The Event System | Qt 4.8*. 2016. URL: https://doc.qt.io/archives/qt-4.8/eventsandfilters.html.

[41] Milosz Malczak. *Distributed Oscilloscope Documentation*. 2019. URL: https://distributed-oscilloscope.readthedocs.io.

[42] Filipe Martins, Anna Kobylinska, and Ulrike Ostler. *Kennzahlen für eine vergleichbare Leistung - Was ist ein Performance Benchmark (wert)? [Translation: Measurements for a comparable performance - What is a performance benchmark (worth)?]* July 2018. URL: https://www.datacenter-insider.de/was-ist-ein-performance-benchmark-wert-a-734687/.

[43] Igor Milovanovic. *Python data visualization cookbook*. Birmingham: Packt Publ., 2013. URL: https://cds.cern.ch/record/1641733.

[44] Alan D. Moore. *Mastering GUI programming with Python: develop impressive cross-platform GUI applications with PyQt*. Packt Publishing, Limited, 2019.

[45] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[46] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993. ISBN: 0125184069. URL: https://www.nngroup.com/books/usability-engineering/.

[47] Allessandro Pasotti. *Python SIP C++ bindings tutorial*. Sept. 2014. URL: https://www.itopen.it/python-sip-c-bindings-experiments/.

[48] Maciej Peryt and Michal Hrabia. *Linac4 Source Autopilot*. Nov. 2019. URL: https://indico.cern.ch/event/857104/contributions/3608676/attachments/1940618/3218085/2019-11-07_BE-CO_TM_Linac4_Source_Autopilot.pdf.

[49] Christian Rentrop and Stephan Augsten. *Performance von Anwendungen testen (engl: Testing the performance of applications)*. Aug. 2017. URL: https://www.dev-insider.de/performance-von-anwendungen-testen-a-634807/.

[50] Selan dos Santos and Ken Brodlie. "Gaining understanding of multivariate and multidimensional data through visualization". In: *Computers & Graphics* 28.3 (2004), pp. 311–325. DOI: 10.1016/j.cag.2004.03.013.

[51] Mukesh Sharma. *Consider Rendering Times While Measuring End-User Performance*. Jan. 2014. URL: https://www.stickyminds.com/article/%20consider-rendering-times-while-measuring-end-user-performance.

[52] Quincy Smith. *The Best Python Data Visualization Libraries*. Nov. 2019. URL: https://www.fusioncharts.com/blog/best-python-data-visualization-libraries/.

[53] *Standard Performance Evaluation Corporation*. URL: http://www.spec.org/.

[54] Robert Taylor. *A beginner's guide to frame rates*. July 2013. URL: https://aframe.com/blog/2013/07/a-beginners-guide-to-frame-rates/.

[55] The Matplotlib development team. *Matplotlib*. 2019. URL: https://matplotlib.org/3.1.1/index.html.

[56] The Matplotlib development team. *Usage Guide*. 2019. URL: https://matplotlib.org/3.1.1/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py.

[57] Trolltech. *Events and Event Filters*. 2005. URL: https://doc.qt.io/archives/3.3/eventsandfilters.html.

[58] Steven Walton. *Grand Theft Auto V Benchmarked: Graphics and CPU Performance*. Apr. 2017. URL: https://www.techspot.com/review/991-gta-5-pc-benchmarks/.

[59] Reinhold P. Weicker. "An overview of common benchmarks". In: *Computer* 23.12 (1990), pp. 65–75. DOI: 10.1109/2.62094.

[60] *Which programming language is fastest?* URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

A. Appendix

A.1. PyQt

A.1.1. PyQt Layout Example Application Source Code

```
import sys
from qtpy import QtWidgets


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args):
        super().__init__(*args)
        self.setup_widgets()
        self.setup_layout()
        self.set_widgets_text()
        self.show()

    def setup_widgets(self):
        # Upper part of the window (horizontally aligned widgets)
        self.central_widget = QtWidgets.QWidget(self)
        self.central_layout = QtWidgets.QVBoxLayout(self.central_widget)
        self.upper_layout = QtWidgets.QHBoxLayout(self.central_widget)
        self.check_box = QtWidgets.QCheckBox(self.central_widget)
        self.label = QtWidgets.QLabel(self.central_widget)
        self.push_button = QtWidgets.QPushButton(self.central_widget)
        # Lower part of the window (tabs)
        self.tab_widget = QtWidgets.QTabWidget(self.central_widget)
        self.content_tab_1 = QtWidgets.QWidget(self.tab_widget)
        self.content_tab_2 = QtWidgets.QWidget(self.tab_widget)
        self.layout_tab_1 = QtWidgets.QGridLayout(self.content_tab_1)
        self.layout_tab_2 = QtWidgets.QGridLayout(self.content_tab_2)
        self.label_tab_1 = QtWidgets.QLabel(self.content_tab_1)
        self.label_tab_2 = QtWidgets.QLabel(self.content_tab_2)

    def setup_layout(self):
        # Set size and central widget
        self.resize(360, 200)
        self.setCentralWidget(self.central_widget)
        # Fill upper half of the window
        self.central_layout.addLayout(self.upper_layout)
        self.upper_layout.addWidget(self.check_box)
```

```
        self.upper_layout.addWidget(self.label)
        self.upper_layout.addWidget(self.push_button)
        self.central_layout.addWidget(self.tab_widget)
        # Fill lower half of the window
        self.tab_widget.addTab(self.content_tab_1, "")
        self.tab_widget.addTab(self.content_tab_2, "")
        self.layout_tab_1.addWidget(self.label_tab_1)
        self.layout_tab_2.addWidget(self.label_tab_2)
        self.tab_widget.setCurrentIndex(0)

    def set_widgets_text(self):
        self.setWindowTitle("Example Application")
        # Set texts in the upper half of the window
        self.check_box.setText("CheckBox")
        self.label.setText("TextLabel")
        self.push_button.setText("PushButton")
        # Set texts in the lower half of the window
        self.tab_widget.setTabText(0, "Tab 1")
        self.tab_widget.setTabText(1, "Tab 2")
        self.label_tab_1.setText("This is the content of Tab I")
        self.label_tab_2.setText("This is the content of Tab II")


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    _ = MainWindow()
    sys.exit(app.exec())
```

Listing A.1: Source code for the layout example in 2.5

A.1.2. Qt Event Sytem Example Application Source Code

```
import sys
from qtpy import QtWidgets, QtGui, QtCore


class ChangingLabel(QtWidgets.QLabel):

    def mousePressEvent(self, e: QtGui.QMouseEvent):
        font: QtGui.QFont = self.font()
        if e.button() == QtCore.Qt.RightButton:
            font.setPointSize(font.pointSize() - 1)
        elif e.button() == QtCore.Qt.LeftButton:
            font.setPointSize(font.pointSize() + 1)
        self.setFont(font)


class LabelMouseClickFilter(QtCore.QObject):

    def eventFilter(self, o: QtCore.QObject, e: QtCore.QEvent):
        """Filter out all Mouse Clicks, single or double"""
        intercept = [QtCore.QEvent.MouseButtonDblClick,
                     QtCore.QEvent.MouseButtonPress]
        if e.type() in intercept and isinstance(o, QtWidgets.QLabel):
            return True
        return super().eventFilter(o, e)


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, app: QtWidgets.QApplication):
        super().__init__()
        self._app = app
        self.setWindowTitle("Event Demo")
        cw = QtWidgets.QWidget()
        self.setCentralWidget(cw)
        cl = QtWidgets.QGridLayout()
        cw.setLayout(cl)
        self._filter = LabelMouseClickFilter(parent=app)
        self._label_1 = ChangingLabel("I react to Mouse Clicks!")
        self._label_2 = ChangingLabel("I react to Mouse Clicks as well!")
        self._checkbox = QtWidgets.QCheckBox("Install Event Filter")
        self._button = QtWidgets.QPushButton("Send Left Click to Labels")
        self._checkbox.stateChanged.connect(self._install_filter)
        self._button.pressed.connect(self._send)
        cl.addWidget(self._label_1, 0, 0)
        cl.addWidget(self._label_2, 0, 1)
        cl.addWidget(self._checkbox, 1, 0)
        cl.addWidget(self._button, 1, 1)
        self.show()
```

```
49
50      def _install_filter(self, checked: int):
51          if checked:
52              self._app.installEventFilter(self._filter)
53          else:
54              self._app.removeEventFilter(self._filter)
55
56      def _send(self):
57          for label in [self._label_1, self._label_2]:
58              event = QtGui.QMouseEvent(
59                  QtCore.QEvent.MouseButtonPress,
60                  QtCore.QPoint(0.0, 0.0),
61                  QtCore.QPoint(0.0, 0.0),
62                  QtCore.QPoint(0.0, 0.0),
63                  QtCore.Qt.LeftButton,
64                  QtCore.Qt.LeftButton,
65                  QtCore.Qt.NoModifier
66              )
67              QtCore.QCoreApplication.postEvent(label, event)
68
69  if __name__ == "__main__":
70      app = QtWidgets.QApplication(sys.argv)
71      event_filter = LabelMouseClickFilter(parent=app)
72      window = MainWindow(app)
73      sys.exit(app.exec())
```

Listing A.2: Source code of an example window demonstrating qts event system

A.2. PyQtGraph

A.2.1. PyQtGraph PlotWidget Components Overview

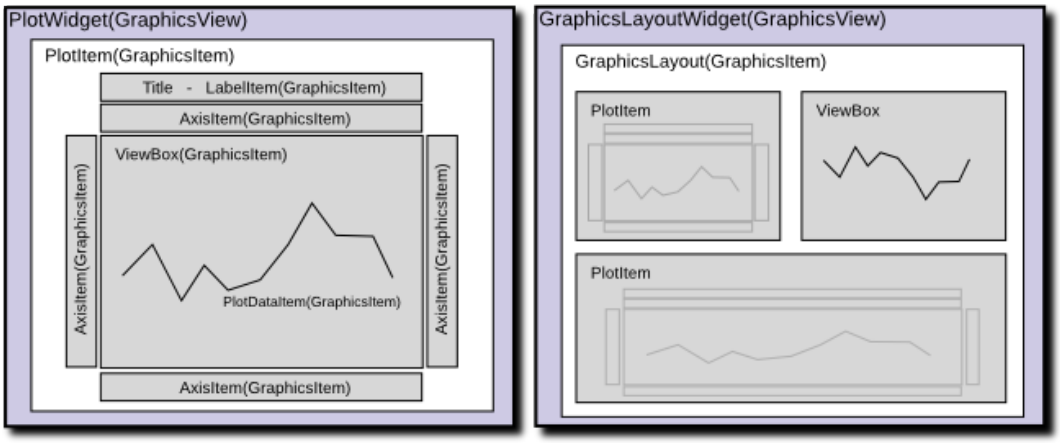


Figure A.1.: Different components of a PyQtGraph plot.
Quelle: http://www.pyqtgraph.org/documentation/plotting.html

A.2.2. PyQtGraph PlotWidget Example Application Source Code

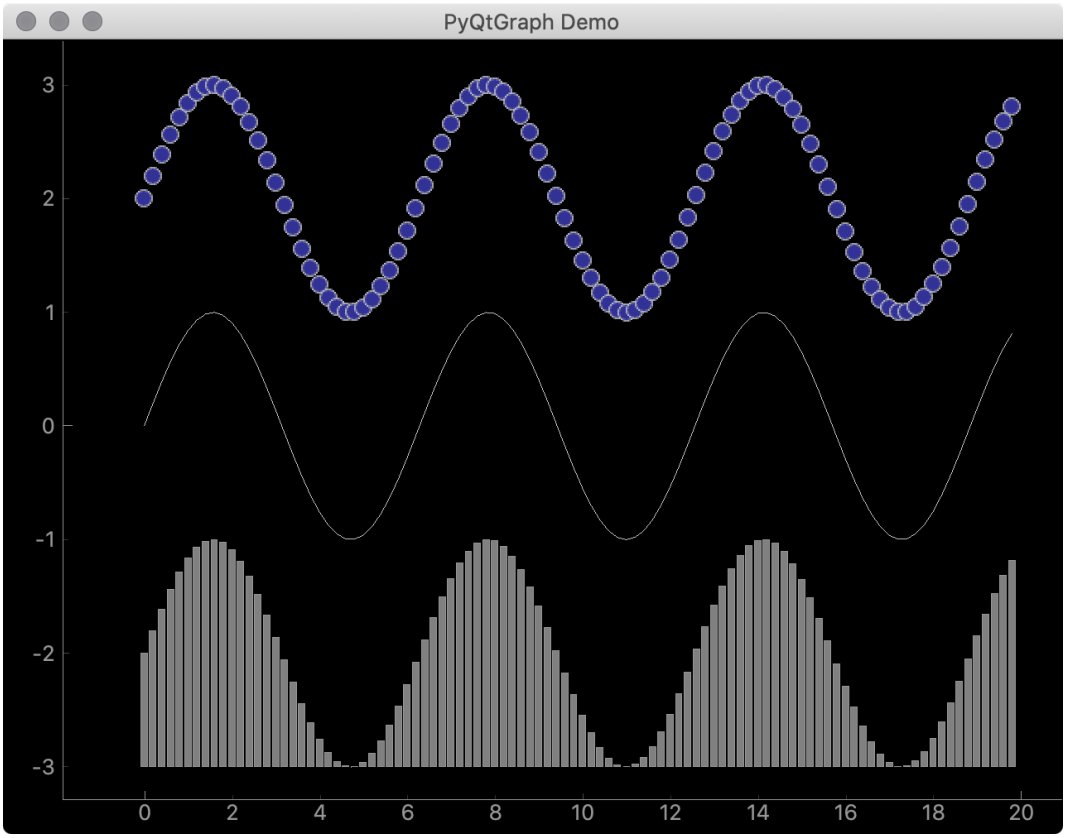


Figure A.2.: Qt Window containing a PyQtGraph plot.

A.3. Matplotlib

A.3.1. Matplotlib Figure Components Overview

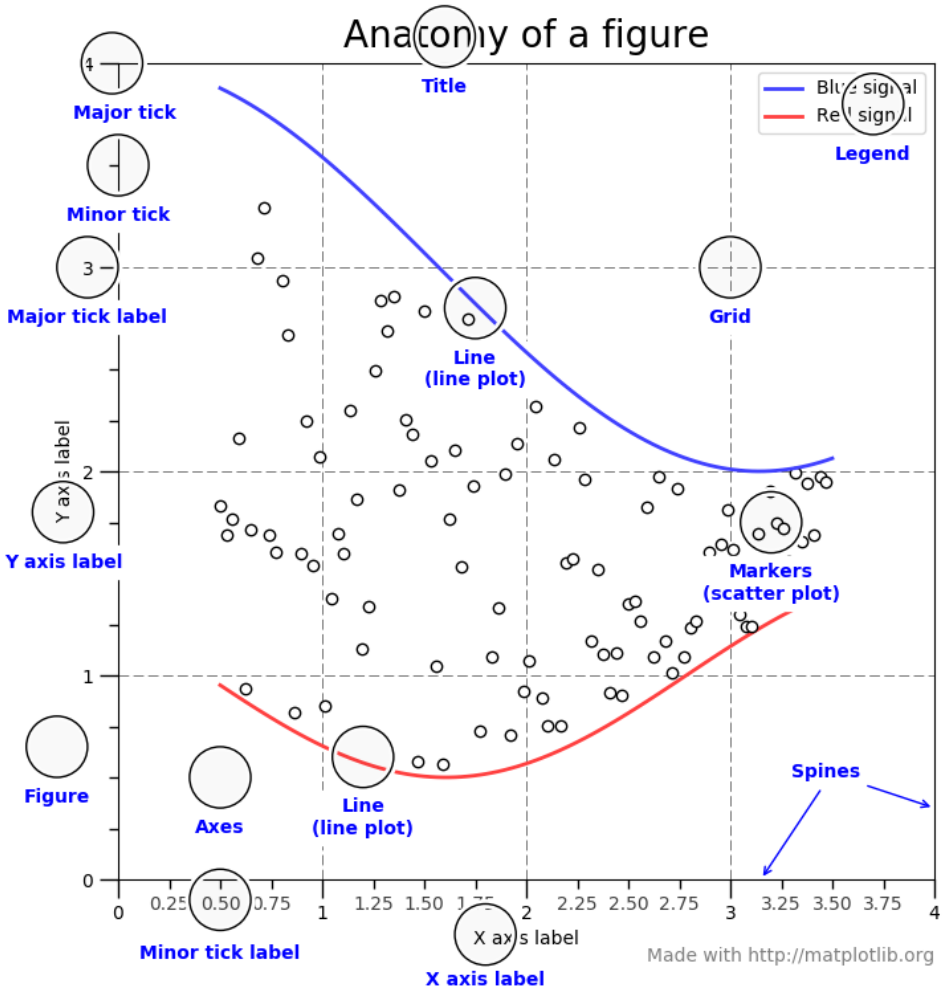


Figure A.3.: Different components of a Matplotlib plot.
Quelle: https://matplotlib.org/3.1.1/tutorials/introductory/usage.html

A.3.2. Matplotlib Figure Example Application Source Code

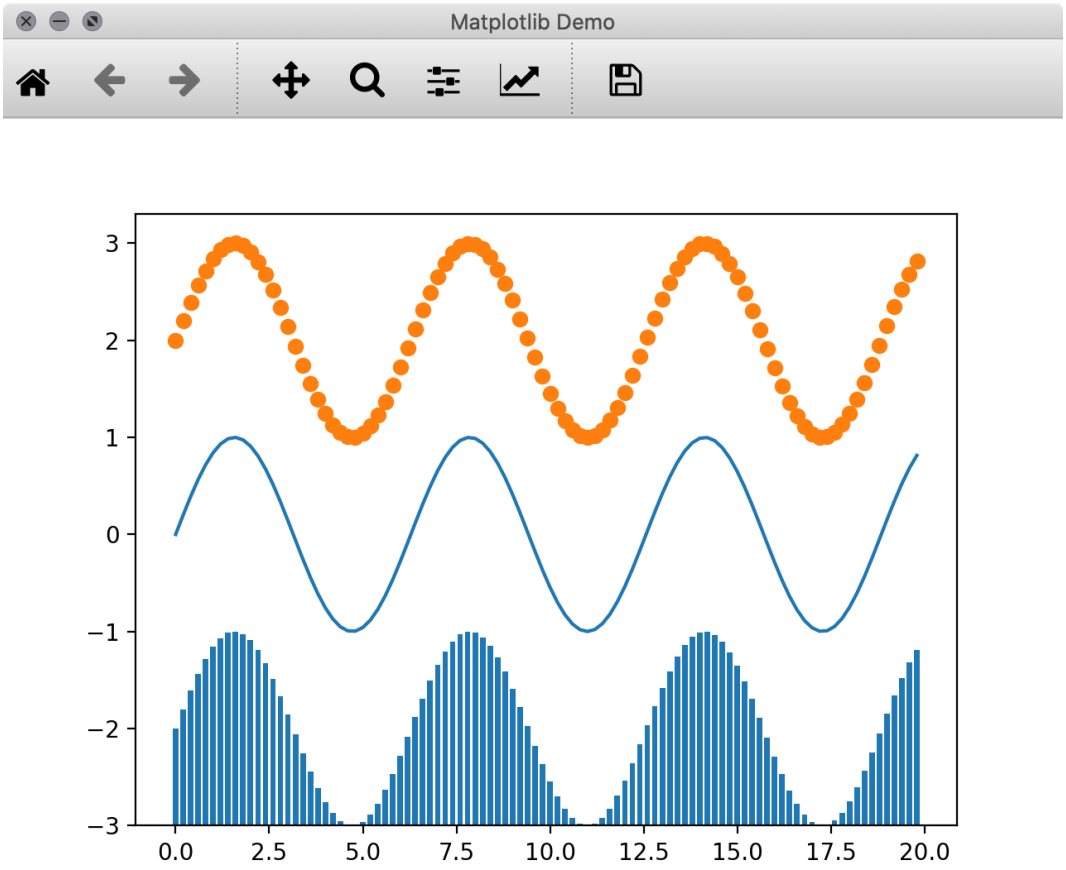


Figure A.4.: Qt Window containing a Matplotlib plot.

A.4. Evaluation

A.4.1. Delta Times for 1,000 Points

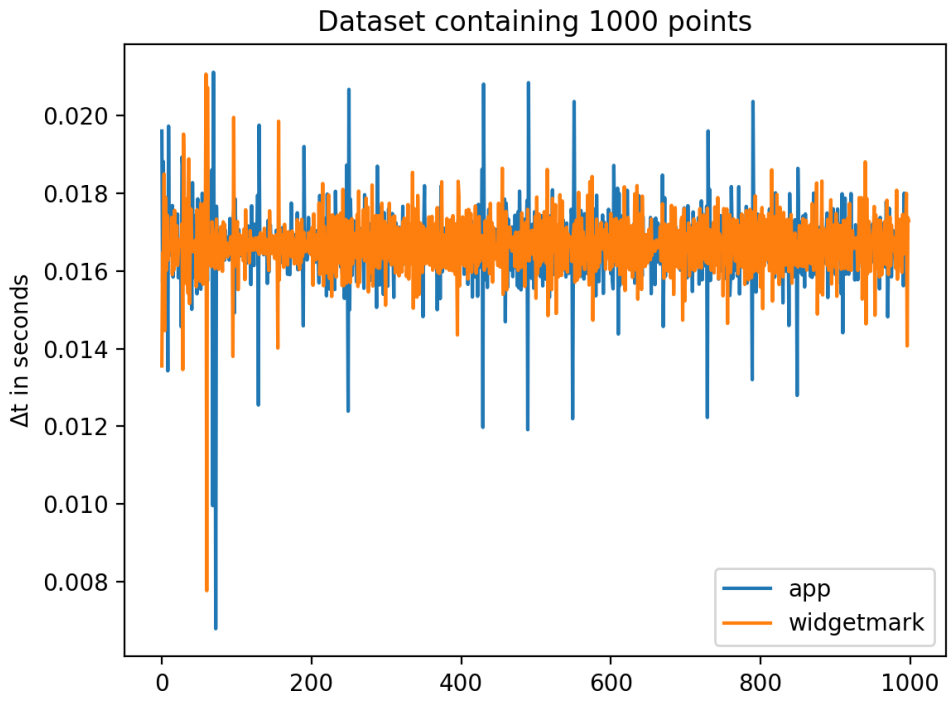


Figure A.5.: Recorded Delta Times for a 1,000 point data set

A.4.2. Delta Times for 10,000 Points

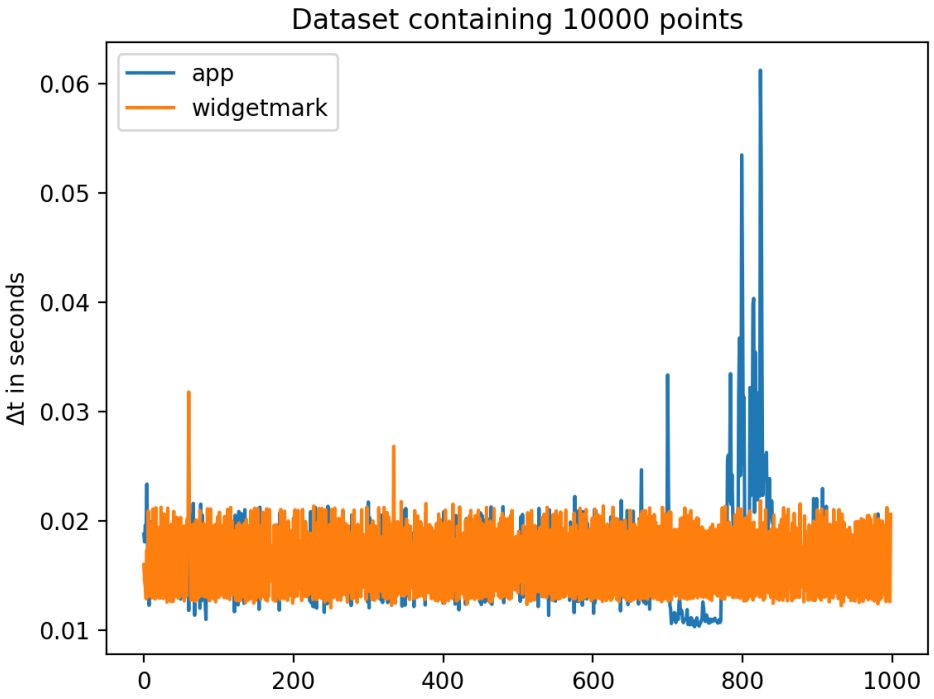


Figure A.6.: Recorded Delta Times for a 10,000 point data set

A.4.3. Delta Times for 50,000 Points

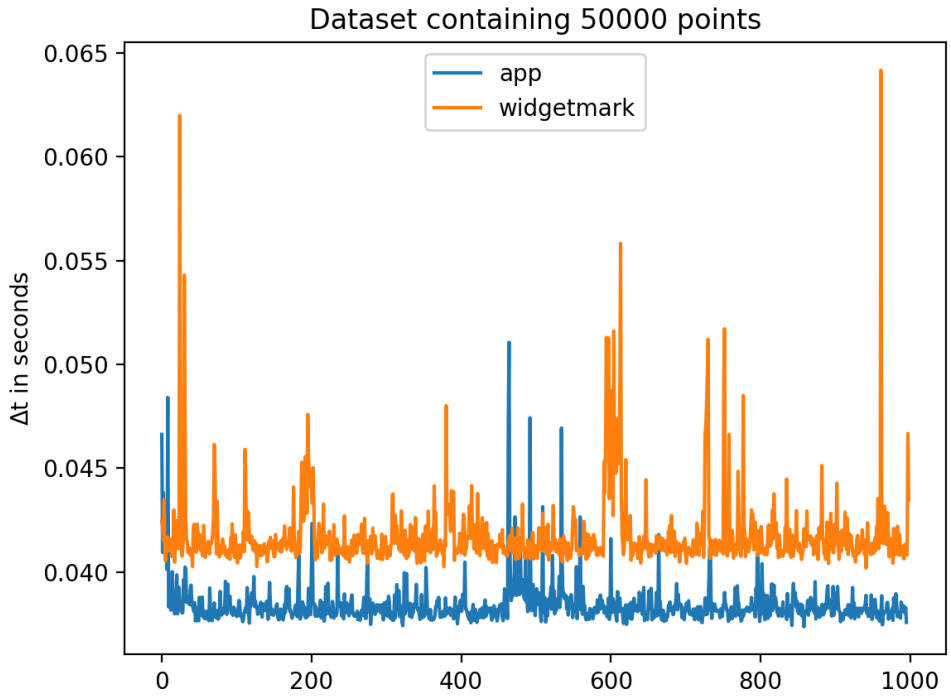


Figure A.7.: Recorded Delta Times for a 50,000 point data set

A.4.4. Delta Times for 100,000 Points

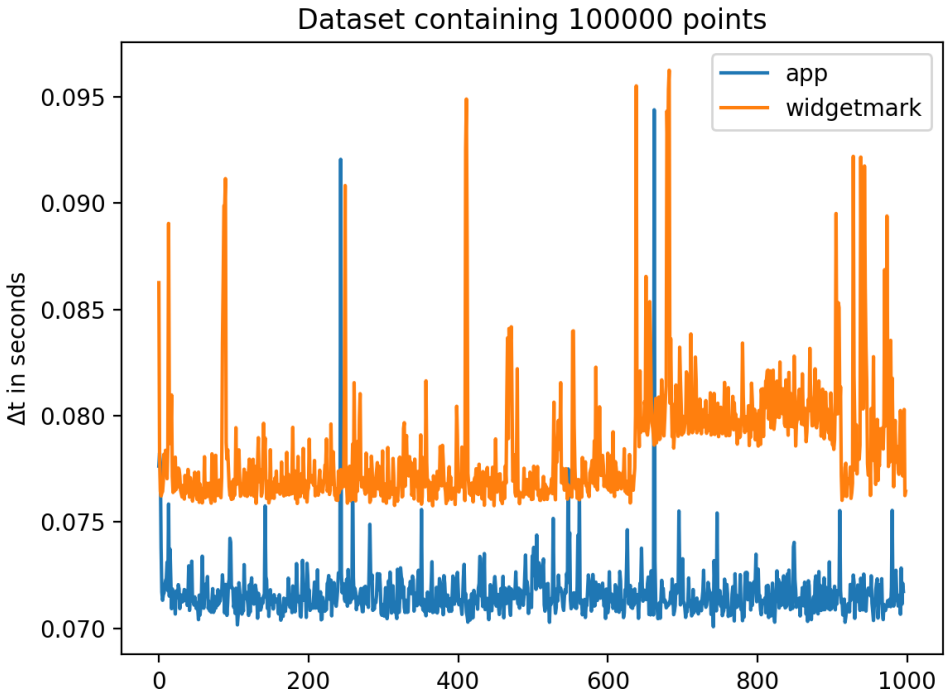


Figure A.8.: Recorded Delta Times for a 100,000 point data set

A.4.5. PyQt Comparison Application Source Code

```
from collections import namedtuple
from pyqtgraph.Qt import QtGui, QtCore, QtWidgets
import sys
import numpy as np
import pyqtgraph as pg
from time import time
import signal
from pdb import set_trace
import sys


DATASET_SIZE = 100000
"""Length of the data set used for this application"""

REPEAT_COUNTER = 1000
"""How often should the plot be redrawn"""


class PyQtGraphWindow(QtWidgets.QMainWindow):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        if len(sys.argv) > 1:
            try:
                ds = int(sys.argv[1])
            except:
                ds = DATASET_SIZE
        else:
            ds = DATASET_SIZE

        self._x = np.linspace(0, 10, ds)
        self._y = np.random.normal(size=(50, ds))
        self._counter = 0
        self._times = []

        self._plot = pg.PlotWidget()
        self._curve = self._plot.plot()
        self._timer = QtCore.QTimer()
        self._timer.timeout.connect(self._update)
        self._timer.start(0)

        self.setCentralWidget(self._plot)
        self.setWindowTitle("PyQtGraph Evaluation Application")
        self.resize(800, 600)
        self._plot.setRange(xRange=(0, 10), yRange=(-5, 5))
        self.show()

```

```
    def _update(self):
        x = self._x
        y = self._y[self._counter % 50]
        self._curve.setData(x=x, y=y)
        self._counter += 1
        self._times.append(time())
        if self._counter >= REPEAT_COUNTER:
            self._timer.stop()
            self.results()

    def results(self):
        fps = "%0.2f" % self.fps
        self._plot.setTitle(f"Results: {fps} FPS")

    @property
    def fps(self) -> float:
        """The current frames per second from all the recorded data"""
        dts = []
        for i in range(len(self._times) - 2):
            dt = self._times[i + 1] - self._times[i]
            dts.append(dt)
        # np.save(f"eval_app_{DATASET_SIZE}", np.array(dts))
        return 1 / (sum(dts) / len(dts))


def main():
    app = QtWidgets.QApplication(sys.argv)
    window = PyQtGraphWindow()
    return_value = app.exec()
    print(window.fps)
    sys.exit(return_value)


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

Listing A.3: PyQt comparison application used in the evaluation of the framework's accuracy

A.4.6. Evaluation Use Case Source Code

```python
import widgetmark
import numpy as np
import pyqtgraph as pg


class Eval(widgetmark.UseCase):

    backend = widgetmark.GuiBackend.QT
    goal = 30.0
    minimum = 20.0
    tolerance = 0.05
    repeat = 1000
    timeout = 100
    parameters = {
        "size": [1000, 10000, 50000, 100000],
    }

    def setup_widget(self):
        pg = widgetmark.PlottingLibraryEnum.PYQTGRAPH
        curve = widgetmark.DataItemType.CURVE
        self._plot = widgetmark.AbstractBasePlot.using(pg)
        self._curve = self._plot.add_item(curve)
        self._x = np.linspace(0, 10, self.size)
        self._y = np.random.normal(size=(50, self.size))
        self._plot.set_range(((0, 10), (-5, 5)))
        return self._plot

    def operate(self):
        y = self._y[self.runtime_context.current_run % 50]
        self._curve.set_data([self._x, y])
```

Listing A.4: Evaluation use case used in the evaluation of the framework's accuracy