



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Benchmarking, analysis, and optimization of Python-based graph library performance

Bachelor Thesis von

Fabian Sorn

an der Fakultät für Informatik und Wirtschaftsinformatik
Fachrichtung Verteilte Systeme (VSYS)

Erstgutachter: Prof. Dr. rer. nat. Christian Zirpins
Zweitgutachter: Prof. B
Zweiter Betreuer: Dipl.-Inform. D

10. November 2019 – 10. March 2020

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

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

PLACE, DATE

.....

(Fabian Sorn)

Zusammenfassung

Deutsche Zusammenfassung

Inhaltsverzeichnis

Zusammenfassung	i
1. Introduction	1
1.1. Motivation	1
1.2. CERN	1
1.3. Beams Controls Applications	2
1.4. Problem	3
1.5. Planned Solution	3
1.6. Structure of this work	4
2. Fundamentals	5
2.1. Data Visualization	5
2.1.1. The development of Data Visualization	5
2.1.2. Data Visualization Pipeline	7
2.1.3. Charting at CERN	8
2.1.4. JDataViewer	10
2.2. Benchmarking	10
2.2.1. Synthetic Hardware Benchmarks	11
2.2.2. Software Benchmarks	12
2.2.3. Application Benchmarks	13
2.2.4. Collection of Benchmarking Criteria	13
2.3. The Qt Application Framework	13
2.3.1. Widgets, Layouts and Widget Hierarchy	14
2.3.2. Signals and Slots	15
2.3.3. The Event System	19
2.3.4. Python bindings	22
3. Use Cases	25
3.1. Digital Oscilloscope for BE-CO-HT	25
3.2. Line Charts in section BE-OP-LHC	25
3.3. Use case in Linac 4 Source GUI	25
3.4. Performance Metrics from Use Cases	26
4. Design and Implementation of a Benchmark Framework	29
4.1. Python Graph Libraries	29
4.1.1. Matplotlib	29
4.1.2. PyQtGraph	30

4.2.	Analysis	33
4.2.1.	Describing Performance of GUI systems	33
4.2.2.	Use Cases	35
4.3.	Design	36
4.3.1.	Benchmark Execution	36
4.3.2.	Use Case Definition	42
4.3.3.	User Interface	43
4.4.	Implementation	46
4.4.1.	Project Structure	46
4.4.2.	Subpackage widgetmark.base	47
4.4.3.	Subpackage widgetmark.qt	52
4.4.4.	Subpackage widgetmark.cli	56
4.5.	Use Case Implementation	57
5.	Performance optimization	59
5.1.	Optimizing line graph performance	59
5.1.1.	GPU accellerated Rendering	59
5.2.	Optimizing scatter plot performance	59
5.2.1.	Incremental data updates	59
6.	Evaluation	61
6.1.	Benchmark changes to Line Graph	61
6.2.	Benchmark changes to Scatter Plot	61
7.	Conclusion	63
7.1.	Summary	63
7.2.	Outlook	63
	Literatur	65
A.	Anhang	69
A.1.	PyQt	69
A.2.	PyQtGraph	73
A.3.	Matplotlib	77

Abbildungsverzeichnis

2.1.	Jacques Barbeu-Dubourg’s Scroll of History	6
2.2.	Transformations and Resulting Data of the Data Visualization Pipeline .	7
2.3.	CERN Control Center	9
2.4.	LEIR Vistar in the Cern Control Center (CCC)	10
2.5.	Layouts and Widgets in a Qt Application Window	16
2.6.	Communication between objects through Signal and Slot Connections .	17
3.1.	Screenshot of the Linac4 Source Gui	26
4.1.	Schema displaying different layers in a single plot.	32
4.2.	Sequence diagram showing the communication between the Command Line Interface and the Framework.	38
4.3.	Sequence diagram showing the creation of the benchmarking window and the executor.	39
4.4.	Sequence diagram showing the actual benchmarking execution on the window by the executor in more detail.	40
4.5.	Class diagram depicting all important classes involved in the Benchmarking Framework.	41
4.6.	Class diagram depicting the plot abstraction interface and two subclasses for the python plotting libraries PyQtGraph and Matplotlib.	42
4.7.	Widgetmark project structure	46
A.1.	Different items involved in a PlotWidget.	73
A.2.	Window containing a plot created with PyQtGraph.	74
A.3.	Initialization of the Live Plotting Architecture for PyQtGraph.	75
A.4.	Update Propagation in the Live Plotting Architecture for PyQtGraph. . .	76
A.5.	Different items involved in a Figure.	77
A.6.	Window containing plot created with Matplotlib.	78

Tabellenverzeichnis

1. Introduction

The following chapter will provide an introduction into this work. First the setting in which the work is done, will be described, starting with the organisation followed by the team. Afterwards the fundamental problem and the goal of this work will be explained, rounded up by an overview about 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. A popular tool 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 most complex and astonishing machines, the Large Hadron Collider (LHC) as well as the birth place of the World Wide Web (WWW), on which we rely on daily everywhere around the world. [8] 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 at in the organisation every day [15].

CERN's roots can be traced back to the 1940's, when a hand full of scientists saw the needs for Europe to advance its role in the scientific world by hosting its 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, organisation and foundations to achieve its goals to expand the boundaries of human knowledge. [11]

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 which have their own purposes. The Beams Department (BE) is responsible for developing software and hardware instruments for the accelerator complex. [7]

The LHC's task is to produce and accelerate two beams of charged particles, travelling 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 collision. The particles detectors then can record the results of the collisions in great detail for later analysis. The operation of the LHC is under one roof, the CCC. [13, 17]

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 behaviour. The development of this control system for the accelerator complex is done by the Controls Group (BE-CO) which is part of BE. [16] 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 it is, to provide software solutions for the many tasks of the control system. One of these products are Graphical User Interfaces (GUIs), that are vital tools for the operators' work. A GUI application allow to monitor the current state of the machine and react to occurring problems by altering the setting of these machines. 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 own 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 [40]. The comparison of offered 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 [45]. 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 each's performance. For more complex operations, like visualizing data, this inevitably raises the question, how we can implement such a measurement and what metrics describe the libraries 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

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 use comparable results to objectively judge performance, but also help us to find and improve slow operations. Afterwards, we will implement potential improvements for found performance deficits. To evaluate our framework, we will run the same benchmarks, but now based on our changed implementation and compare the results to the original implementation.

1.6. Structure of this work

In the beginning of this work, 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 next chapter 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. Following that, we will have a look at the basics of benchmarking as well as already existing benchmarking solutions, from which we can derive concepts, we can use for our own implementation. The next chapter then guides through the design and implementation of a benchmark framework and benchmarking suite, which allows us to run our graph library of choice against the collected use cases. Following that, we can use the benchmarks detailed results, to plan ways to remove performance bottle-necks in the implementation. Finally the exact same benchmarks will be run again, to objectively judge, which impact our changes had on the tested operations.

2. Fundamentals

This chapter will give an introduction into the fundamental topics that are relevant for 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 which describes in general the functionality and steps of modern data visualization frameworks and the different types of data visualization applications. Finally the chapter 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 chapter we will move our focus on the topic of performance benchmarking. First, we will explore the different types of benchmarks that exist on the market and followed by an introduction, how UI performance can be benchmarked. Since the focus of our benchmarking framework will focus on applications using Qt and its python bindings, we will have a look at its fundamental principles in the third section, which we will have to know about, if we want to understand the performance of graph widgets written for it. The fourth and last section of this chapter will give an introduction into PyQtGraph, the plotting library of choice in BE-CO-APS and the adaptations we are developing for it in BE-CO-APS.

2.1. Data Visualization

The following chapter gives a brief introduction into the history and development of data visualization, presents a model which 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. Representing data in a visual way has its root in the earliest of humanities history, where drawings on cave walls were a way to share stories, which's 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, whitespaces 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. [24]

Visualization does not only help us in understanding static data, but also discovering trends in its development over a period of time. In 1753, Jacques Barbeu-Dubourg created a graph displaying a large historical time line, called the *Carne Chronographique*. The graph displays events from a total time span 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 exact 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. [21]



Abbildung 2.1.: Jacques Barbeu-Dubourg's Scroll of History
Quelle: <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. [20]

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 which describes the functionality of such libraries, is the Data Visualization Pipeline. [25]

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.

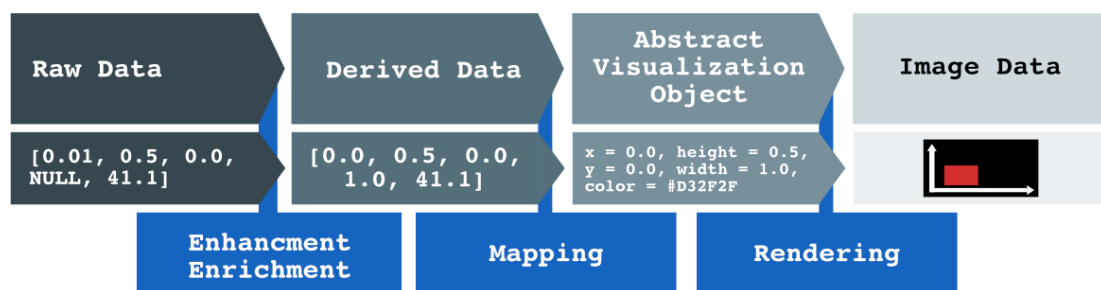


Abbildung 2.2.: Transformations and Resulting Data of the Data Visualization Pipeline

The starting point of the pipeline is the raw data acquisitioned 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 for such an attribute could be a color, which is calculated from one of the dimensions in derived data. An example for 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 an 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.

[25, 39]

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 period of time, so the main priority of performance is, that the storage system can keep up with the high band-width 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 visualisation software systems are 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 visualisation 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 the input parameters. The success of the changes can be actively monitored while it is running.

[25]

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 a 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 analysed and visualized offline. One example for such a display is a

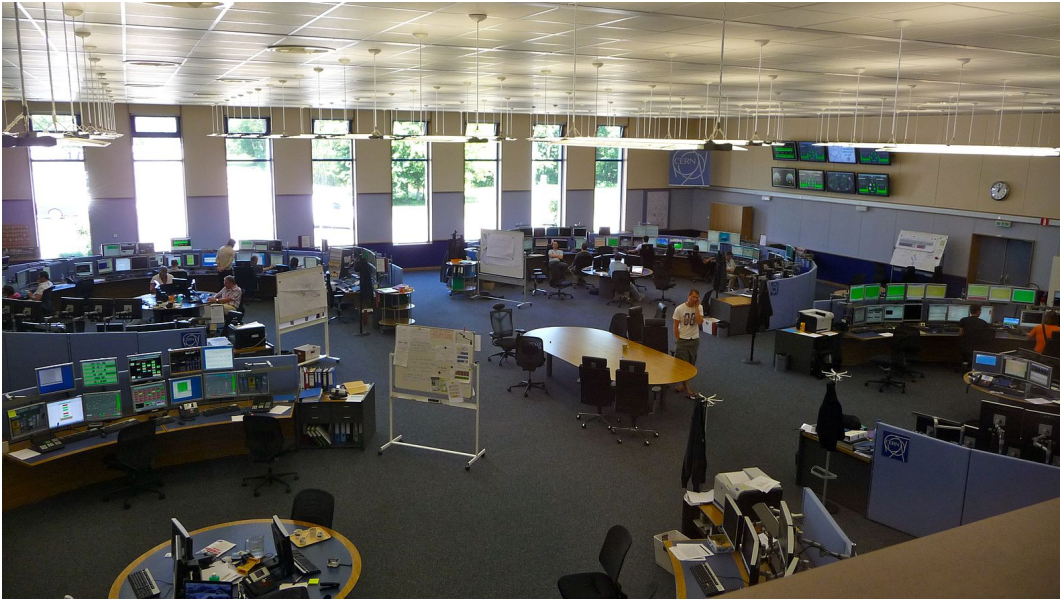


Abbildung 2.3.: CERN Control Center

three dimensional view of recorded collisions in the ATLAS Control room, which can be followed live online, if the detector is running.

[12, 9, 18, 19]

Implementations of the second and third type 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, which's 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 monitor different parts of the accelerator complex. These parts are the Technical Infrastructure (TI), Proton Synchrotron (PS), Super 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.

[5]

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.

[16]

One Vistar used as 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

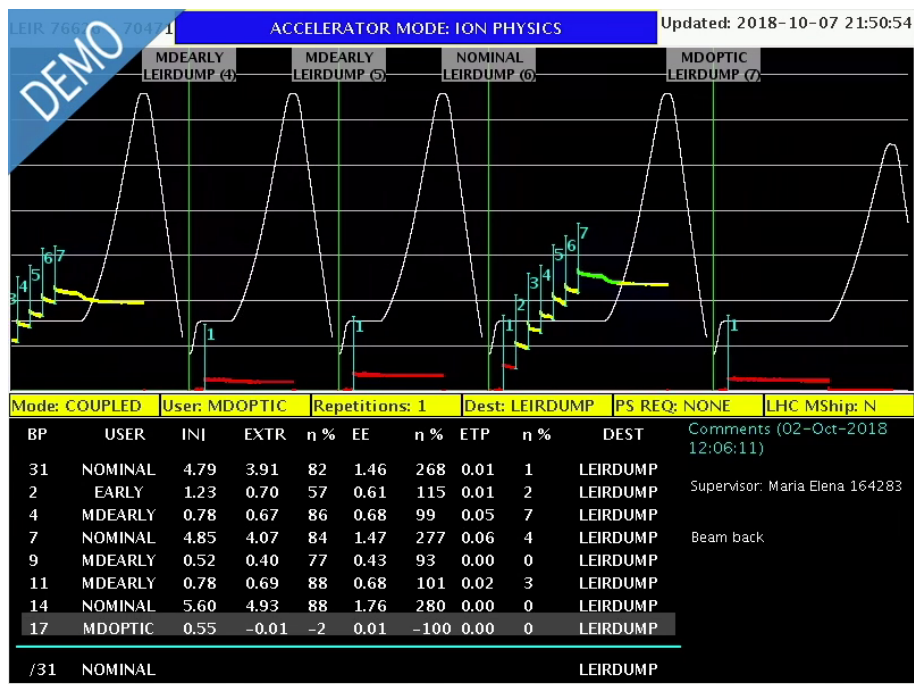


Abbildung 2.4.: LEIR Vistar in the CCC

LHC. Its Vistar contains a graph at the top displaying the injection of particles, the energy level and the different cycles.

[14]

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 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. Goal for its PyQt predecessor is to offer similar features and performance.

[27]

2.2. Benchmarking

The main advantage that 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, cofounder 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 a large amount of to be processed data.

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 are able to acquire is often bound to a specific budget. Since more computing power will inevitably mean a higher count of transistors, cores and clockspeeds, the price of hardware will raise with its computing power. A less costly approach is the optimization of the software which is used to perform a task. 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 analyse performance objectively, to take a sound decision.

[36]

Benchmarking is a procedure which allows us to objectively compare the performance of a system or process, to find out, which work loads 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 a tested hardware achieved in common benchmarking applications to give the potential customer a way to compare different offerings on the market.

[2]

This chapter will give an overview about different approaches on performance evaluation in the world of computing.

2.2.1. Synthetic Hardware Benchmarks

Hardware benchmarks can range from simple and synthetic sequences of operations to much more complex application like benchmarks that try to test a component's performance 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 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 GPUs. The GeForce RTX 2080, at the time of writing very recent GPU from the GPU manufacturer NVIDIA, can theoretically reach up to 10.6 TeraFLOPS (TFLOPS) or $10.6 \cdot 10^{12}$ FLOPS.

[6]

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 Algol 60. The goal of Whetstone was to test a hardware's performance without relying on any hardware specific instructions by replacing 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 were 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 computers speed in solving operations on matrices. The result of the benchmarks were published in FLOPS.

[45]

A more modern suite of benchmarking applications is provided by SPEC. Spec is a non profit-organization which 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 an interesting measurement.

[41]

Although it plays a very large role in the performance of an application on a specific system, 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 be bad. Because of this another approach to benchmarking is Software Benchmarking.

2.2.2. Software Benchmarks

Next to benchmarking hardware, we can also measure the performance of software. Benchmarking software is executing the same task in different implementations on the same hardware and compare the recorded results. One example of 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 languages in implementing this algorithm. 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 projects official webpage. 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 are not capable of displaying the entire performance capability of the language. [46]

2.2.3. Application Benchmarks

For most modern hardware, measurements like FLOPS are not very helpful, when choosing hardware or the speed of a certain programming language when working with binary trees, when choosing a software to perform a certain task. The tasks of which's performance we actually 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, which give us a much more realistic image of the performance of hardware in our later use cases. Depending on the use case, there are many offerings in Application Benchmarking Software on the market.

Especially in journalism about computer hardware you can find a very wide spread usage of application benchmarks to describe the performance of new released hardware products and how they will perform compared to older models. Very common are benchmarks, that will use mostly CPU or the GPU to perform tasks that are very close to the use cases of the customers of the hardware products.

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 criterias.

- 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.

- 2) To give us a realistic view on a machines performance, the Benchmark should also be representative to 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 users 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 are widely distributed. If benchmarking results are only available for one certain piece of hardware, 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 into 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 base for GUI applications, we will take the same decision for our implementation. This chapter will give a general introduction in the creation of GUIs 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 licences 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 2.3.3.

The code example 2.1 shows these mentioned steps in a running Qt Application written in Python Code. The window it is producing is not yet containing 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

It is worth mentioning two additional details in regards to example 2.1. 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 command line. The second one is, that we pass 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 UI building blocks, like labels, checkboxes, comboboxes and more can be found and 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 layouting 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 ones with additional functionality. Other options for layouting a window is grouping widgets into containers 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 as seen visible in depiction 2.5. On the right side of the depiction you can see the widget and layout types contained in the result window visible on the left. The source code for this window is appended in A.1. [35]

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. In the following section we will have a look, how we can attach logic to the user interface components.

2.3.2. Signals and Slots

At this point, all our examples for GUI applications have not yet been very useful, since the interaction with elements does not yet invoke any actions. 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 realise this connection through callbacks. A callback is a pointer to a function, which is invoked, when interaction with a certain user

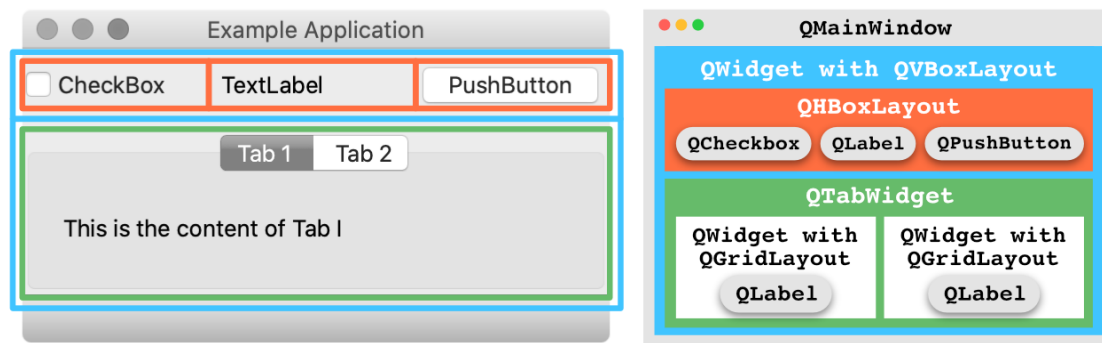


Abbildung 2.5.: Layouts and Widgets in a Qt Application Window

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 from each other, since neither the slots knows, which signals it is connected to, nor the signals knows which slots it is connected with. Compared to callbacks, the Signal and Slots have more Overhead, which makes their execution slower. Compared to any f.e. 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. [32, 35]

Listing 2.2 shows a simple scenario, where the signal of a `QtWidgets.QLineEdit`, is connected to a slot of a `QtWidgets.QLabel`. Both Signal and Slot are already implemented 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.

```
1 import sys
2 from qtpy import QtWidgets
3
4
```

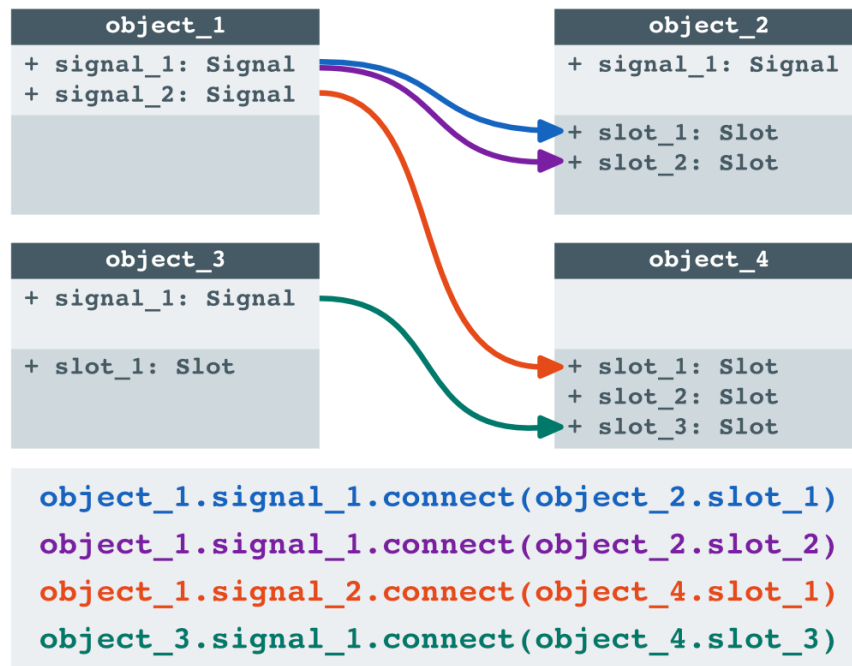


Abbildung 2.6.: Communication between objects through Signal and Slot Connections

```

5 class MainWindow(QMainWindow):
6
7     def __init__(self, *args):
8         super().__init__(*args)
9         self.label = QtWidgets.QLabel()
10        self.edit = QtWidgets.QLineEdit()
11        # Connect Text-Changed-Signal to the Set-Text-Slot
12        self.edit.textChanged.connect(self.label.setText)
13        self.setup_layout()
14        self.show()
15
16    def setup_layout(self):
17        """Take the widgets and put them into a layout"""
18        central_widget = QtWidgets.QWidget()
19        central_layout = QtWidgets.QVBoxLayout()
20        central_widget.setLayout(central_layout)
21        self.setCentralWidget(central_widget)
22        central_layout.addWidget(self.label)
23        central_layout.addWidget(self.edit)
24        self.setWindowTitle("Signals and Slots Demo")
25
26
27 if __name__ == "__main__":
28     app = QtWidgets.QApplication(sys.argv)
29     _ = MainWindow()

```

```
30 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. This is possible as well. For this we will extend our example with another Label from our own type `WatchLabel` derived from `QtWidgets.QLabel`, which offers a slot `set_time`, that takes a float value, interprets it as a timestamp and displays it. To update this label, we will introduce another object `WatchTick` derived from `QtCore.QObject`, which sends a signal 60 times per second containing the current timestamp. 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.

```
1 import time
2 from qtpy import QtWidgets, QtCore, QtGui
3
4
5 class WatchLabel(QtWidgets.QLabel):
6
7     def __init__(self):
8         QtWidgets.QLabel.__init__(self)
9         font = QtGui.QFont("Courier")
10        font.setWeight(QtGui.QFont.ExtraBold)
11        self.setFont(font)
12
13        @QtCore.Slot(float)
14        def set_time(self, timestamp: float):
15            """Takes timestamp and displays it in a readable format."""
16            self.setText(str(time.ctime(timestamp)))
17
18
19 class WatchTick(QtCore.QObject):
20
21     sig_time = QtCore.Signal(float)
22
23     def __init__(self):
24         QtCore.QObject.__init__(self)
25         self.timer = QtCore.QTimer()
26         self.timer.timeout.connect(self.emit_time)
27
28     def emit_time(self):
29         self.sig_time.emit(time.time())
30
31     def start(self):
32         self.timer.start(1000/60)
```

Listing 2.3: Defining a custom signal and slot

As in listing 2.2, we can simply connect the new slot with the signal. After starting the timer, the signal will be emitted around 60 times per second, which will update the label, which then shows the current time.

```

1     self.watch_tick = WatchTick()
2     self.watch = WatchLabel()
3     # Connect Text-Changed-Signal to the Set-Text-Slot
4     self.edit.textChanged.connect(self.label.setText)
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, which, when executing in the main GUI Thread, would block all interaction with the user interface. 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 for that 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, which 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 object are living.

[31]

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 which has 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 also 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 a the line `app.exec()` in them. This call will start Qt's Event Loop and will be blocking until the application terminates. 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 window or adding content to it are for example such Events. In Qt all these events are packaged into 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 this 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 can appear 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 either through `QtCore.QCoreApplication.sendEvent()` or `QtCore.QCoreApplication.postEvent()`.

The first will lead to immediate delivering 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 change its state in any way when receiving Mouse Press Events.

```
1 from qtpy import QtCore, QtGui, QtWidgets
2
3 class ChangingLabelOne(QtWidgets.QLabel):
4     pass
5
6 class MainWindow(QtWidgets.QMainWindow):
7
8     def __init__(self):
9         super().__init__()
10        widget = ChangingLabelOne("Test")
11        self.setCentralWidget(widget)
12        event = QtGui.QMouseEvent(
13            QtCore.QEvent.MouseButtonPress,
14            QtCore.QPoint(0.0, 0.0),
15            QtCore.QPoint(0.0, 0.0),
16            QtCore.QPoint(0.0, 0.0),
17            QtCore.Qt.LeftButton,
18            QtCore.Qt.LeftButton,
19            QtCore.Qt.NoModifier
20        )
21        QtCore.QCoreApplication.postEvent(
22            widget, event
23        )
```



```
24 self.show()
```

Listing 2.5: Positing a `MouseEvent` to a Widget

When it is time to process an Event, Qt delivers them to widgets, so they can react to them if they want. The type of the Event does also influence in which way it is delivered. If a event is not accepted by a child widget, it is propagated to its parent widget, where this principle repeats. If an event is accepted by a widget and how the widget reacts to it, is defined in a Widgets Event Handler.

Widgets have two options when defining their behavior on the delivery of Events. The first option to handle arriving Events is `QtCore.QObject.event()`, which is the general event handler for all kinds 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 accpets Mouse Click Events. If the widget detects a Mouse Click Event, it accpets it and increases it 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 thorough 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 before 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 thorough 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 widgets, 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 are simply classes derived from `QtCore.QObject`, which implement the `QtCore.QObject.eventFilter()` protected method. 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 `QtWidgets.QApplication` instance as you can see as an example in listing 2.8. As before, we have to return `True` to prevent Qt from further propagating the event to other widgets.

```
1 import sys
2 from qtpy import QtCore, QtGui, QtWidgets
3
4 class MouseClickFilter(QtCore.QObject):
5
6     def eventFilter(self, _: QtCore.QObject, e: QtCore.QEvent):
7         """Filter out all Mouse Clicks, single or double"""
8         intercept = [QtCore.QEvent.MouseButtonDblClick,
9                     QtCore.QEvent.MouseButtonPress]
10        if e.type() in intercept:
11            print("Filter Mouse Click.")
12            return True
13        return super().eventFilter(_, e)
14
15 if __name__ == "__main__":
16     app = QtWidgets.QApplication(sys.argv)
17     app.installEventFilter(MouseClickFilter(parent=app))
18     win = QtWidgets.QMainWindow()
19     win.show()
20     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 which demonstrate these three concepts of event creation, even 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, a button sending a mouse press events to both labels and a checkbox which installs an EventFilter for Mouse Clicks on Labels and removes it again when unchecked. [33, 3, 44]

2.3.4. Python bindings

Even though we do now have a basic understanding about the fundamental principles 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 Applications 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 version of PyQt. [28]

Qt is a big Framework containing over one thousand classes, which have to be accessible through Python to reflect the complete set of Features. It is generally possible to write these bindings by hand, but a lot of work. To automate the creation of Python Bindings, Riverbank has developed the tool SIP, which can generate such Python bindings from interface header files with similar syntax to C++ header files. These files define what classes and methods should be exported and how to call the module it should be exported into. Additionally it is possible to define the translation between Python and C++ Objects. SIP then takes this header file 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. [29, 37, 22]

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 on the other hand has its own Garbage collection mechanism, which can delete objects if their parent object is deleted or if they do not have any parent object. This deletion does not influence the existence of the object on the Python side in any way, which makes it possible trying to access a deleted C++ objects 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++ 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, as seen in listing 2.9, to delete the object and its children.

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

Listing 2.9: Example for properly deleting a QWidget

Another caveat when working with PyQt is, when working with different Threads. When working for example 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 like complex calculations, it won't be executed parallel on systems which would support true parallel execution. The reason for this is Python's Global Interpreter Lock (GIL). The GIL was introduced

to python for memory management reasons to keep track of object references. 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, 23]

Since PyQt5 is not the only available Python binding for Qt, we will use the Python package `qtpy` for all our code, which is a simple pure Python abstraction layer, which will delegate all imports to whatever Python binding for Qt is installed. This has the big advantage, that the code, we developed with PyQt5, will run also in environments which have for example the Qt binding PySide or PyQt4 installed. Listing 2.10 shows those two options.

```
1 # Direct usage of a Python binding, will fail if environment doesn't use PyQt5
2 from PyQt5 import QtCore, QtGui, QtWidgets
3 # Using qtpy for Qt related imports, will work with PyQt and PySide
4 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 over different Use Cases that we will later use for the evaluation of the Plotting Libraries. All of these are different scenarios from Teams at CERN, that are looking into PyQt as an option to implement different GUI applications and need plots in their applications.

3.1. Digital Oscilloscope for BE-CO-HT

The first Use Case we will investigate is coming from the Hardware and Timing Section (BE-CO-HT), who are planning to implement a GUI application showing a digital oscilloscope. For this they are interested in displaying a Line Graph in their application, which is showing up to eight curves with up to 100.000 points per curve. The goal for this graph would be an update rate of 25 updates per second.

3.2. Line Charts in section BE-OP-LHC

The second Use Case we will investigate is coming from the Operations Group LHC (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. Use case in Linac 4 Source GUI

The third Use Case we will investigate is coming from BE-CO-APS. For this, multiple ScatterPlots should be displayed. The GUI Application will contain 4 different scatter plots, each containing up to 3 data sets, which each contain 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 tasks it is to boost negative hydrogen ions to high energies. The GUI will allow monitoring and manipulation of different device settings. Image 3.1 shows a screenshot of an early version of the application. In the upper right part of the window are two graphs containing multiple scatter plots. Both graphs are implemented with the python graph library pyqtgraph, which we will explore in more detail in section 4.1.2. [10, 38]



Abbildung 3.1.: Screenshot of the Linac4 Source Gui

3.4. Performance Metrics from Use Cases

In this section we will collect metrics we can use to define a plotting libraries performance based on the insights we gained into the usage of these libraries from the use cases in this chapter. All of these Use Cases have in common, that they are displaying live data, which will be delivered with a certain frequency. If a plotting widget needs a very long time to redraw, this might result in updates piling up without the widget having time to redraw, if an update is always leading to a redraw. Long redraw cycles are not only a The most important metric we have to measure is without a doubt, how long the operations following a data update take. When collecting those times and calculating the average from it, we can make realistic predictions, if a widget will handle a certain update load under certain circumstances.

Even for applications containing graphs, which are not updated regularly, the time it takes the graph to redraw is very important for the interaction with the graphs, since it requires the graph to redraw as well. Slow redraw times will lead in these applications to stuttering use interfaces.

If a Use Case reveals a performance issue, it is important to investigate, where the source of the performance issue is rooted. Profiling is a very helpful procedure to decide, where improvements are needed most. A profile lists very detailed information about which

functions consumed how much time in total, how often it has been called and how long a single call did take. With these informations, we can investigate, where we can change code to increase performance.

4. Design and Implementation of a Benchmark Framework

In this chapter, the findings from the previous chapters will be combined to design and implement a benchmarking framework, which allows the user to benchmark charting operations, he has defined himself.

4.1. Python Graph Libraries

When benchmarking different Python Graph libraries, we will focus our attention on two contenders for the implementation, which .

4.1.1. Matplotlib

Matplotlib is probably 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. [42, 26]

Matplotlibs 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 represents the minimum and maximum data limit for each dimension of the data. An Axes object can be personalised through axis labels and a title. In Matplotlib terminology, Artists are everything which is visible in a Figure. This includes, Labels, Line 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.

Matplotlib is built for many different use cases. While showing data in GUI applications is one of them, it can be used for other use cases, like generating plots as images for publications. This is achieved by Matplotlib supporting different Backends. A Backend is the system, which allows you to use Matplotlib in these different use cases. All available Backends can be divided in interactive and hardcopy ones. An example for a interactive backend is in our case PyQt5, but other Frameworks like Tkinter or PyGTK are supported. Non interactive backends are for creating image files in different file formats like PNG, SVG, PDF and more. [43, 34]

Listing 4.1 shows the creation of a window containing a plot representing a sinus curve in different ways. 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.6.

```
1 import numpy as np
2 from matplotlib.backends.backend_qt5agg import (
3     FigureCanvasQTAgg as FigureCanvas,
4     NavigationToolbar2QT as NavigationToolbar,
5 )
6 from matplotlib.figure import Figure
7
8 class MatplotlibWindow(QtWidgets.QMainWindow):
9
10     def __init__(self, **kwargs):
11         super().__init__(**kwargs)
12         # Data
13         x = np.array(range(100)) * 0.2
14         y = np.sin(x)
15         # Plot + Navigation Bar
16         self._canvas = FigureCanvas(Figure())
17         self._plot = self._canvas.figure.subplots()
18         self._toolbar = NavigationToolbar(canvas=self._canvas, parent=self)
19         # Data Visualization
20         curve = self._plot.plot(x, y, "-")
21         scatter = self._plot.plot(x, y + 2, "o")
22         bars = self._plot.bar(x=x, height=y + 1, bottom=-3, width=0.15)
23         # Setup Window Content
24         layout = QtWidgets.QVBoxLayout()
25         layout.setSpacing(0)
26         layout.setContentsMargins(0, 0, 0, 0)
27         layout.addWidget(self._toolbar)
28         layout.addWidget(self._canvas)
29         central_widget = QtWidgets.QWidget()
30         central_widget.setLayout(layout)
31         self.setCentralWidget(central_widget)
32         self.setWindowTitle("Matplotlib Demo")
```

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

4.1.2. PyQtGraph

PyQtGraph is a pure python plotting library. While not offering as many features as other Python plotting libraries like Matplotlib, PyQtGraph promises to offer much better Plotting performance. PyQtGraph 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 offers widgets for 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 plots with different data visualization. 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 its fast performance and the possibility for the user to interact and transform the scene through for example zooming or rotation. [30]

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`. Since it is derived from `QtWidgets.QWidget`, it is very simple to add to a window. 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 `PlotItem` contains different components of the plot, including a `pyqtgraph.ViewBox`, which is the area data is visualized in, a `pyqtgraph.AxisItem`, which is representing the View Range of the View Box and a Title. Items which are actually representing a dataset are added to the View Box. For this, PyQtGraph is offering different types of representations like a `pyqtgraph.PlotDataItem` for Scatter Plots and Line Graphs or `pyqtgraph.BarGraphItem` for representing data in a bar graph.

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 example from 4.1.1. For the Line Graph and Scatter Plot we can use `pyqtgraph.PlotDataItem`. For the Bar Graph we create a `pyqtgraph.BarGraphItem`. Each item can then be added to the plot. The resulting window can be seen in A.2. [4]

```

1 import numpy as np
2 from qtpy import QtWidgets
3 import pyqtgraph
4
5 class PyQtGraphWindow(QtWidgets.QMainWindow):
6
7     def __init__(self, **kwargs):
8         super().__init__(**kwargs)
9         # Data
10        x = np.array(range(100)) * 0.2
11        y = np.sin(x)
12        # Plot
13        self._plot = pyqtgraph.PlotWidget()
14        # Data Visualization
15        curve = pyqtgraph.PlotDataItem(x=x, y=y)
16        scatter = pyqtgraph.PlotDataItem(x=x, y=y + 2, pen=None, symbol="o")
17        bars = pyqtgraph.BarGraphItem(x=x, height=y + 1, y0=-3, width = 0.15)
18        self._plot.addItem(curve)
19        self._plot.addItem(scatter)
20        self._plot.addItem(bars)
21        # Setup window content
22        self.setCentralWidget(self._plot)
23        self.setWindowTitle("PyQtGraph Demo")
24        self.show()

```

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

4.1.2.1. Extensions for PyQtGraph from BE-CO-APS

PyQtGraph has already seen adaption in projects at CERN, which needed fast plotting performance. To provide a more convenient and faster development experience, BE-CO-APS is providing an extended version of PyQtGraph mainly adapted for usage in Runtime Monitoring Applications. This extension offers first of all the additions of important features missing in PyQtGraph and as second a convenient live data plotting layer.

One feature which PyQtGraph is lacking, is the ability to plot two or more datasets on different y-axes. This feature is especially useful, when you are viewing and comparing data sets, whose values are in vastly different ranges on the y-axis. One example where this feature was heavily missing, was the use case described in section 3.3. As seen in the screenshot 3.1 the plot in the upper right corner contains two axes on the left and right side. The left one is showing a range from approximately minus 35 to 15, while the right one shows a range from minus 15000 to 35000. To keep the plot interactive, the different data sets are displayed on different areas, which are `pyqtgraph.ViewBox`. The shown view range is displayed by an `pyqtgraph.AxisItem`. In the extension a layer describes one of these viewboxes together with its axis. Schema 4.1 shows how multiple layers are stacked on top of each other to build a single plot. The big advantage is, that each layer can be moved individually by the user of the application.

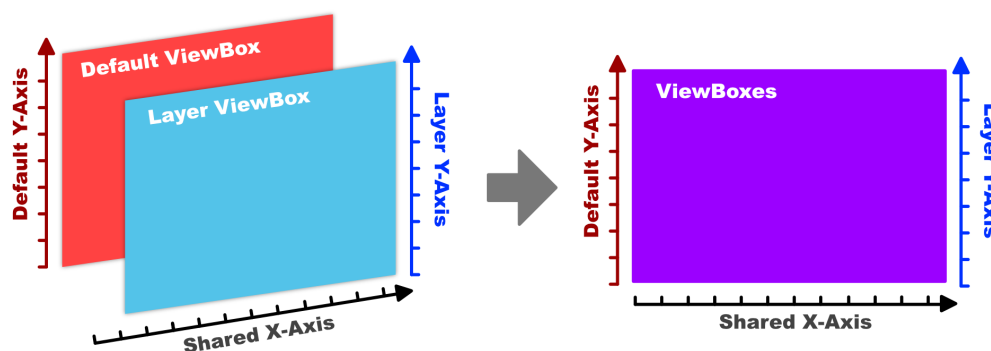


Abbildung 4.1.: Schema displaying different layers in a single plot.

The other addition made to PyQtGraph is a convenience layer on top of different types of data visualizations, which allow much easier connection of processes emitting data to each data visualization item. Pure PyQtGraph does not have any way of incrementally feeding data into a plot, it only allows replacing the entire data set. As an example we assume a user, who wants to display the last hour of data coming from a process, which is continuously emitting new data with a frequency of 1 Hz. With pure PyQtGraph the user will have to handle saving and creating a subset of data, he is interested in all by himself. Additionally, data coming from a data logging system, which contains data with older time stamps compared to the ones from the live data, these points will have to be sorted into the existing array of data. The reason for that is, that PyQtGraph draws the connection between points by their position in the passed array, not by the x values

provided. Additionally creating a subset of the last one hour of data for an unsorted data set will be a much harder task compared to a sorted data set. The convenience layer for PyQtGraph does remove these tasks from the users responsibilities by allowing him to define a time range he is actually interested in and the only feed new data to the graph without having to store or sort it.

To achieve this, three different components have been introduced: the Update Source, the Data Model and the View. The Update Source is responsible for connecting to a device or process whose data is supposed to be visualized. The data model is responsible for storing the data and keeping it sorted. The View is responsible for displaying the data from the datamodel. Depiction A.3 shows the initialization process of these different layers. The View is a single data is not the entire plot, but a single data visualization object, like curves or bar graphs.

Depiction A.4 is showing the communication as soon as new data is available in the process. The Update Source receives the new data from the process. How this data is transferred is specific to the protocols used for communication as well as the type of process connected to. From there the Update Source emits the new data thorough signals to the data model, which sorts it in to the right position in the internal buffer. Afterwards the view is informed through signals, which can then take the subset of data, which it wants to display.

Both, pure PyQtGraph as well as the extended versions will be included in our benchmarks, especially to investigate potential performance hits introduced by the new live data plotting architecture.

4.2. Analysis

After having a closer look at the two plotting libraries we will focus on, this chapter will focus on the design of the benchmarking framework, which we can use to investigate performance of both in the different use cases.

This section will focus on the conceptual design of such a framework, while the following section 4.4 will focus on actual implementation.

4.2.1. Describing Performance of GUI systems

From a Usability Engineering perspective, rendering performance is a vital aspect, since it greatly impacts a GUIs response time. The general advice on system's response times is 100 milliseconds, which is roughly the boundary up to which a systems feels like it is instantly responding to input. Delays over one second can already interrupt the thought process of the user, since he is clearly noticing the delay. Delays over ten seconds on the other hand require feedback from the system to signal the user it is performing long lasting tasks, since the user often already wants to perform other tasks.

An often used measurment for describing a system's rendering performance is the Framerate, which describes the frequency, with which images appear on a screen. This size originates from the movie industry, where the standard framerate 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 Framrates 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 Framrates are desirable, since they will not only enhance the visual experience, but will give the player a strategic advantage in a competitive environment. A popular demonstration of the increased clarity of high framrates is the Blur Buster's UFO Motion Blur Test website, which shows the same scene of fast moving cartoon ufo in different Framrates (<https://www.testufo.com>).

In many Hardware Benchmarks, Frames per Second 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. In Video Games for example, this information allows us to find out, how much an object in the displayed scene should have moved since the last displayed frame. As a result, figure movement stays consistent, even under different load scenarios.

From the recorded timing information, we can calculate the Framrate f from a set of m recorded time stamps t_n as follows.

$$f = \frac{1}{\frac{\sum_{n=0}^{m-1} t_{n+1} - t_n}{m-1}}$$

While Framrates already provides very important information from a user's perspective, it does not yet help improving those from a developers perspective. Profilers are software tools which are able to measure different aspects of performance, like execution time, memory management, garbage collection and more. The type of profiler we will mainly focus on is deterministic CPU utilization profilers, collects information about time spent in functions, the origin and amount of calls to these functions. With the help of this information, optimization efforts can be directed to where they really can make a difference. Additionally Profilers can not only uncover slow performing fuctions, but also, if functions are more often called as intended by a programs author.

The standard option in Python to profile code, is the c extension cProfile, which allows deterministic profiling of your Python programs. Compared to the profiling profile, which is written in Python, cProfile offers a much lower overhead, which makes it suitable for profiling longer running programs Compared to the profiling profile, which is written in Python, cProfile offers a much lower overhead, which makes it suitable for profiling longer running programs. To provide this information about the execution of a program, cProfile does use hooks for events, which are offered by the Pyhton interpreter. These hooks are callbacks, which allow cProfile to receive and record timing information about every function executed by the Pyhton interpreter. For profile and cProfile, there are two limitations, which you have to be aware of when profiling sotware. The first one is the granularity of the underlying timer used for recording the duration of an operation, which is mostly around 0.001 seconds. The other limitation is the lag between the event being

dispatched and the profiler to get the current time. Since profiling is introducing a certain overhead to the execution of a python program, it is not suitable for benchmarking itself.

4.2.2. Use Cases

One important aspect of designing a framework, is the possible use cases it offers to the user. Users interested in the performance of a graph library in a specific scenario, have a clear idea of their use case. Most higher level descriptions can be broken down in a simple operation or a sequence of operations. Since these Use Cases are the central point of interaction with the Framework, the we will define the following requirements, which have to be met for the definition of Use Cases.

Ease of Use The Framework should handle everything that is not relevant for the Use Case. The user should only define, what is part of the Use Case and nothing else.

Widget and Operation The central items for defining a Use Case are a widget, and an operation which 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 Use Cases take much longer than expected, it should be possible to define a timeout, which will terminate the Use Case execution, if reached.

One framework type, which allows a similar way of defining specific use cases, are Testing Frameworks like unittest or pytest. Since most users are to an extent familiar with their functionality, the interface should conform 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 like other python tools. The interface should conform the user's expectations in the same way with a few additions specific for our needs.

Use Case Execution Similar to Frameworks like pytest, it should be able to execute a full suite of Use Cases or only specific ones.

Meaningful Results After execution, the user should be able to clearly see the results of his Use Cases. This includes the Use Case, its performance requirements, the actually achieved performance and the parameters used in the run.

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

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

4.3. Design

This section will focus on the design decisions for the framework and describing the components involved. The most outer layer of the framework contains two components. The first component is the command line interface, which offers the user the possibility to execute his benchmarking use cases. Additionally the interface offers functionality to display the results gathered by the framework. The second component is the actual framework in the narrow sense, which is responsible for providing the necessary interfaces for the definition of use cases, the functionality for executing use cases, as well as recording the performance metrics that are displayed later. We will refer to the executing part of the framework as the launcher from now on.

4.3.1. Benchmark Execution

Figure 4.2 shows the communication between the user, the command line interface and the launcher for the application case of executing a benchmarking use case from the command line. When the User starts the executable, the a CLI instance is created, which parses the command line arguments and instantiates the launcher. To keep the user interface exchangable, launcher and CLI are strictly separated from each other. After the Launcher is initialized, it will import the use case modules. With the completion of the step, the launcher can be started, which will execute all collected use cases, collect the results and return them to the CLI, where they can be displayed.

As next we will have a more detailed look at the execution of a collection of use cases, which is visualized in depiction 4.3. We want to allow executing a whole suite of use cases similar to testing frameworks allowing us to execute entire test suites. This means that our launcher has to be able to handle multiple use cases, execute them sequentially and save each's record. Additionally, the launcher has to be able to handle parameterized use cases and execute them for every possible parameter combination similar to the parameterize functionalities of many testing framework.

For the execution we will introduce two new components to our framework. The first component is the GUI window, which we will embed the widget in, which we want to operate on. The second one is the executor, which is responsible to execute the defined operation on the window and record timing information and the profile. After our Launcher is initialized and our benchmark files are properly imported, we will execute each use case in all possible parameter combinations. For each Combination we create a benchmarking window, which embeds the widget defined in the use case to display it. The window will then be passed to the executor, which needs the window to execute the operations defined in the use case on it. After the executor is fully initialized, we can run it.

Depiction 4.4 shows this process in more detail. Before each operation, the current time stamp is taken. If a profile is supposed to be created, the profiler is started here as well. Afterwards the operation, which we want to benchmark, is executed on the passed window instance. After it and all of its side effects are completed, the profiler is stopped if necessary. This execution cycle is repeated, until the defined repeat count or the timeout is reached. Afterwards, the timing information and the profile data are can be returned to the launcher.

Figure 4.5 shows the classes from the prior sequence diagrams and their relations to each other. Since Launcher and CLI are clearly separated from each other, it will be easily possible to replace the User Interface without having to change any business logic of the framework.

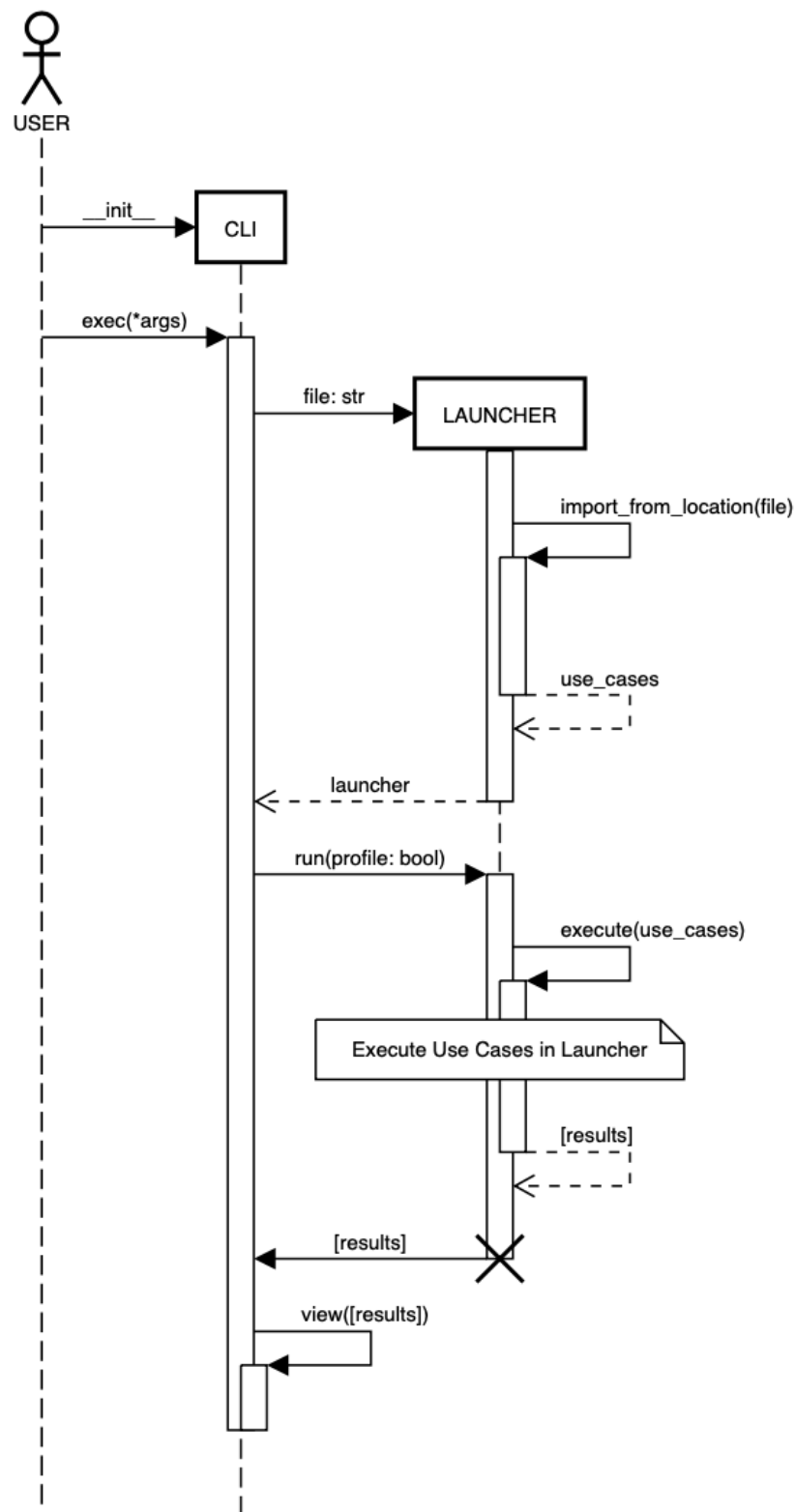


Abbildung 4.2.: Sequence diagram showing the communication between the Command Line Interface and the Framework.

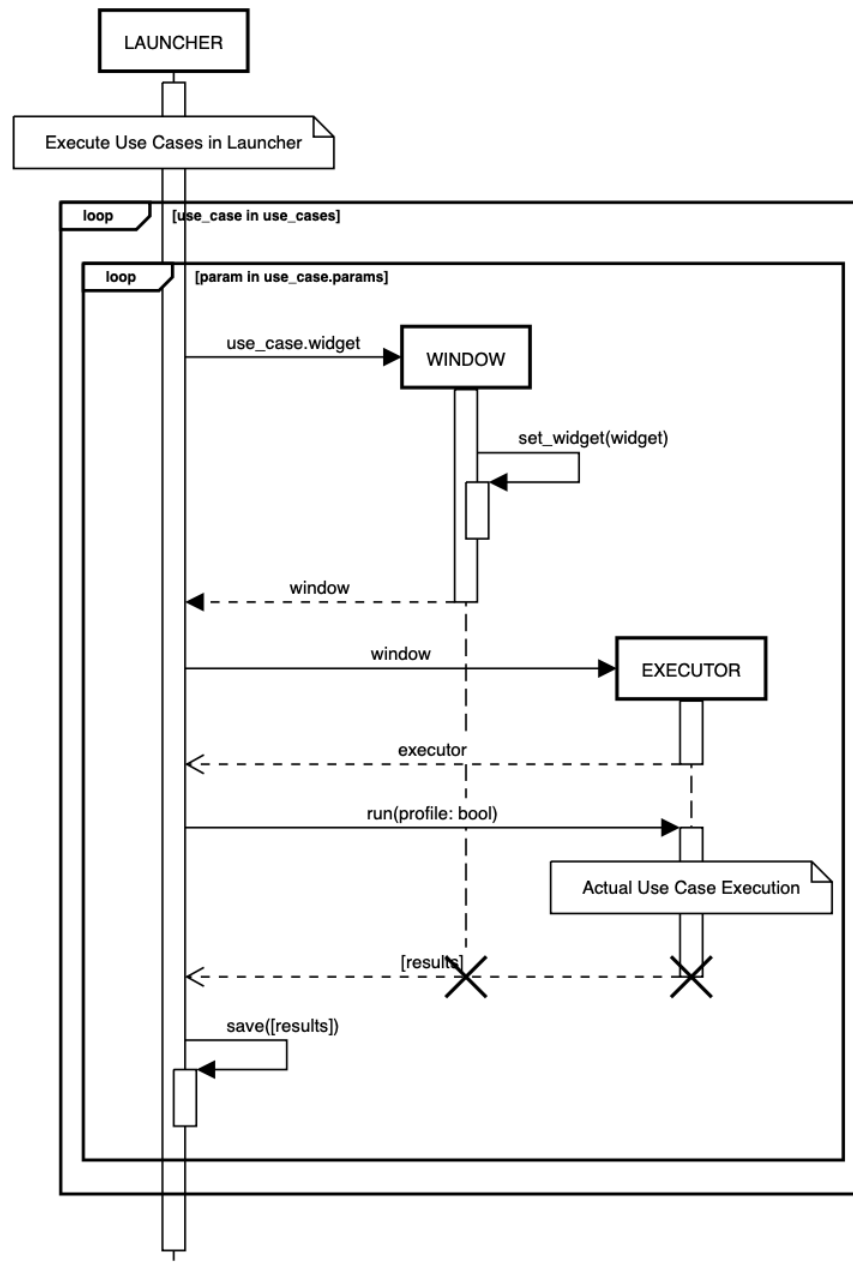


Abbildung 4.3.: Sequence diagram showing the creation of the benchmarking window and the executor.

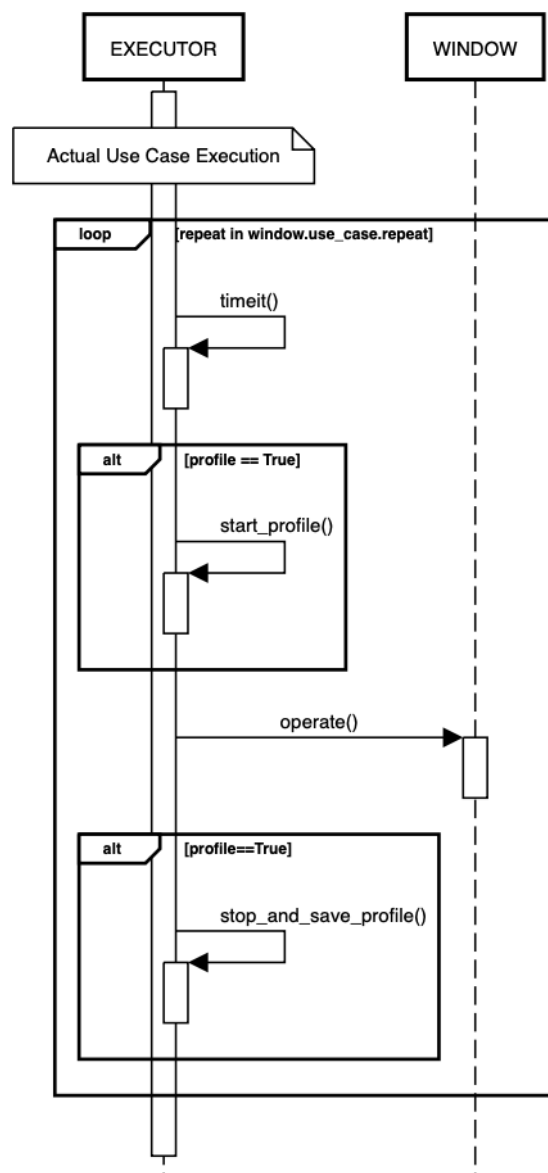


Abbildung 4.4.: Sequence diagram showing the actual benchmarking execution on the window by the executor in more detail.

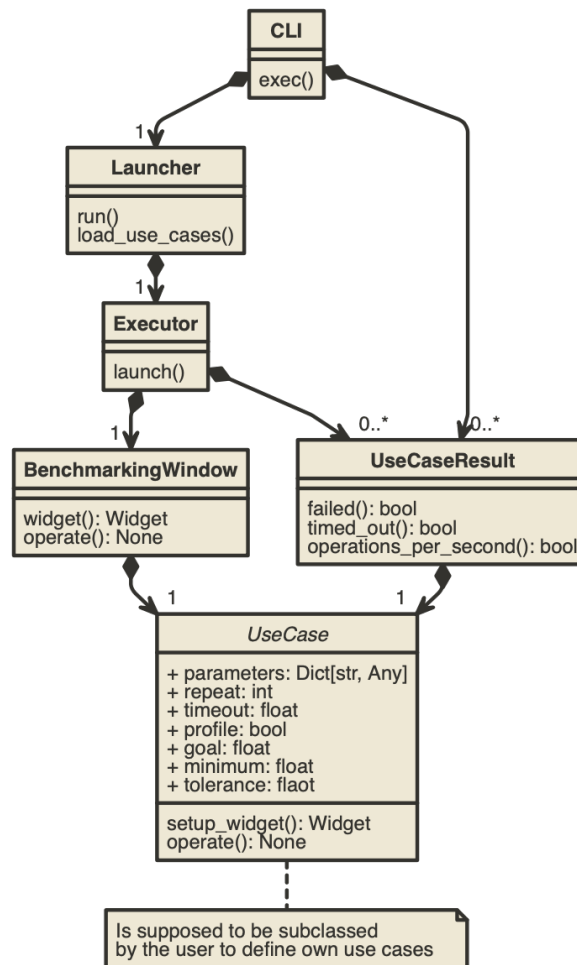


Abbildung 4.5.: Class diagram depicting all important classes involved in the Benchmarking Framework.

4.3.2. Use Case Definition

In this section we will have a closer look at the design of Use Case definition after collecting the requirements for Use Cases in section 4.2.2. To define a Use Case, Users will have to define a class in a python module. To mark the class as a Use Case, it will have to subclass the class `UseCase`. This will allow us to have a clearly defined interface for Use Cases, and makes the discovery of use case classes easier on execution. Figure 4.5 shows the classes methods.

To define plotting benchmarks that are executable with different plotting libraries we will implement a unified interface for them, that can be implemented using different graph libraries. For now we will limit this interface to the functionality of our use cases we need. Such an interface is of course limited as well to the functionality that all libraries used for the implementation offer in common. With using the parameterization functionalities of our Use Cases, we can simply define higher level plotting benchmarks and execute them using different libraries to compare the results from each library.

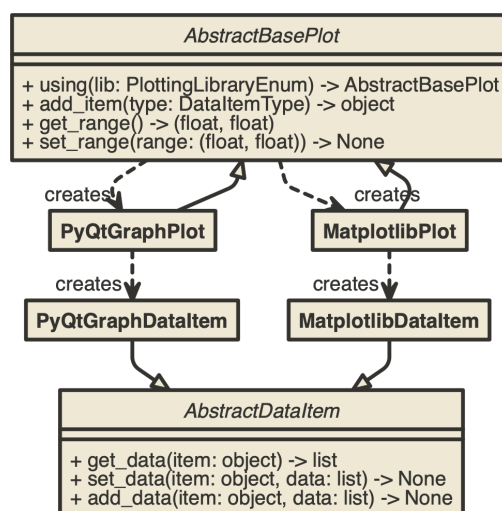


Abbildung 4.6.: Class diagram depicting the plot abstraction interface and two subclasses for the python plotting libraries PyQtGraph and Matplotlib.

Figure 4.6 shows the 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 the current view range, set the current view range and a factory method for creating a plot using a specific plotting library. The two subclasses `PyQtGraphPlot` and `MatplotlibCode` can implement the functionality defined by the base class using each's libraries 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 plot, the two subclasses `PyQtGraphDataItem` and `MatplotlibDataItem` implement this.

Code Example 4.3 shows an example of the interface usage in a standard Qt window. Which library is used for the visualization can be simply controlled by choosing it during the widget's creation using the `AbstractBasePlot` factory method.

```

1 class Window(QtWidgets.QMainWindow):
2
3     def __init__(self, *args, **kwargs):
4         """
5         Changing the library passed to the factory method of AbstractBasePlot
6         allows switching between the libraries used in for the visualization.
7         """
8         super().__init__(*args, **kwargs)
9         # self.plot = AbstractBasePlot.using(PlottingLibraryEnum.PYQTGRAPH)
10        self.plot = AbstractBasePlot.using(PlottingLibraryEnum.MATPLOTLIB)
11        # Add Scatter Plot Data Item to the plot.
12        item = self.plot.add_item(DataItemType.SCATTER)
13        # Set Data of the Scatter Plot
14        x = np.linspace(0, 2 * pi, 20)
15        y = np.sin(x)
16        item.set_data([x, y])
17        # Set visible view range
18        x_range = [0, 2 * pi]
19        y_range = [-1, 1]
20        self.plot.set_range([x_range, y_range])
21        # Add plot to the window
22        self.setCentralWidget(self.plot)

```

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

4.3.3. User Interface

In this section we will have a look at the design of the command line interface of the framework based on the requirements we collected in 4.2.2. Since our command line is supposed to be easily usable for users of other testing frameworks, our command line interface should be kept similar. Listing 4.4 shows how to specify the location of one or multiple use cases from the command line. We will use a specific naming convention to mark files who define use cases. Each file which is supposed to be searched has to start with the prefix `bench_*`. This pattern should be configurable from the command line as well.

```

1 # Print Project structure
2 foo@mbp ~ $: ll my_benchmarks
3   -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot.py
4   -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_image.py
5   -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 b_video.py
6 # Search files recursively from current working directory
7 foo@mbp ~ $: widgetmark
8   Executing bench_plot.py
9   - SinusCurve
10  - ZoomingPlot
11  Executing bench_image.py
12  - HdImage
13 # Search files recursively from current working directory

```

```
14 foo@mbp ~ $: widgetmark .
15     Executing bench_plot.py
16     - SinusCurve
17     - ZoomingPlot
18     Executing bench_image.py
19     - HdImage
20 # Search files recursively from given folder
21 foo@mbp ~ $: widgetmark my_benchmarks
22     Executing bench_plot.py
23     - SinusCurve
24     - ZoomingPlot
25     Executing bench_image.py
26     - HdImage
27 # Search use cases in given file
28 foo@mbp ~ $: widgetmark my_benchmarks/bench_plot.py
29     Executing bench_plot.py
30     - SinusCurve
31     - ZoomingPlot
32 # Execute only single specific use case
33 foo@mbp ~ $: widgetmark my_benchmarks/bench_plot.py::SinusCurve
34     Executing bench_plot.py
35     - SinusCurve
36 # Search for files with different pattern
37 foo@mbp ~ $: widgetmark --pattern "b_*"
38     Executing b_video.py
39     - HdVideo
```

Listing 4.4: Starting benchmarks 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.

```
1 # Start widgetmark and create a profile
2 foo@mbp ~ $: widgetmark --profile --profile-output=profiles .
3     Executing bench_plot.py
4     - SinusCurve
5     - ZoomingPlot
6     Executing bench_image.py
7     - HdImage
8     Saving 3 profile files to ./profiles
9 # Profile outputs
10 foo@mbp ~ $: ll profiles
11 -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot_SinusCurve.profile
12 -rw-r--r--  1 foo  bar   100B  1 Jan 00:00 bench_plot_ZoomingPlot.profile
13 -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 these binary profile files, which listing ?? is showing. Simply printing profile files 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 profile files.

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 additional output when searching for the cause of bugs.

```

1 # Usage explanation for the command line interface
2 foo@mbp ~ $: widgetmark --help
3     usage: widgetmark [-h] [-o PROFILE_FILES_LOCATION] [-p]
4                       [--pattern USE_CASE_FILE_NAME_PATTERN] [--visualize]
5                       [--loglevel LOGGING_MODULE_LEVEL]
6                       [locations [locations ...]]
7
8     A short explanation what widgetmark is.
9
10    Explanation of the individual arguments:
11    ...

```

Listing 4.6: Starting benchmarks and create profiles for them

The CLI output should contain all important information about the found files, the use cases, the performance requirements, the parameters and the actual reached framerate. Listing 4.7 shows these informations on the command line after all found benchmarks were executed.

```

1 # Output after the benchmark execution
2 foo@mbp ~ $: widgetmark .
3 ----- WIDGET-MARK -----
4 > bench_image.py
5 + HdImage          GOAL=30.0, MIN=20.0: ..... 21.2
6 > bench_plot.py
7 + SinusCurve
8   - PYQTGRAPH      GOAL=30.0, MIN=20.0: ..... 56.6
9   - MATPLOTLIB     GOAL=30.0, MIN=20.0: . 1.3
10   - Timed out after 20 seconds.
11 -----
12 Summary (Executed 3 Use Cases)
13 GOAL      1 Use Cases
14 MIN      1 Use Cases
15 NONE      1 Use Cases
16 -----
17 Exceptions 0 Use Cases
18 Timed Out  1 Use Cases
19 -----

```

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

4.4. Implementation

This section will focus on the actual implementation of the framework based on the Analysis from section 4.2, the design from section 4.3 and the Use Cases from ???. As a name we choose *widgetmark*, a composite of widget and benchmark. The language we use is Python in the version 3.6 to use newer language features like typing hint, which will highly improve the usability of the framework.

4.4.1. Project Structure

The first part of the implementation is setting up our project in Python, which figure 4.7 is showing. 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 a introduction into the project, *setup.cfg* for tooling configuration and *setup.py* for dependency definition and installation instructions.

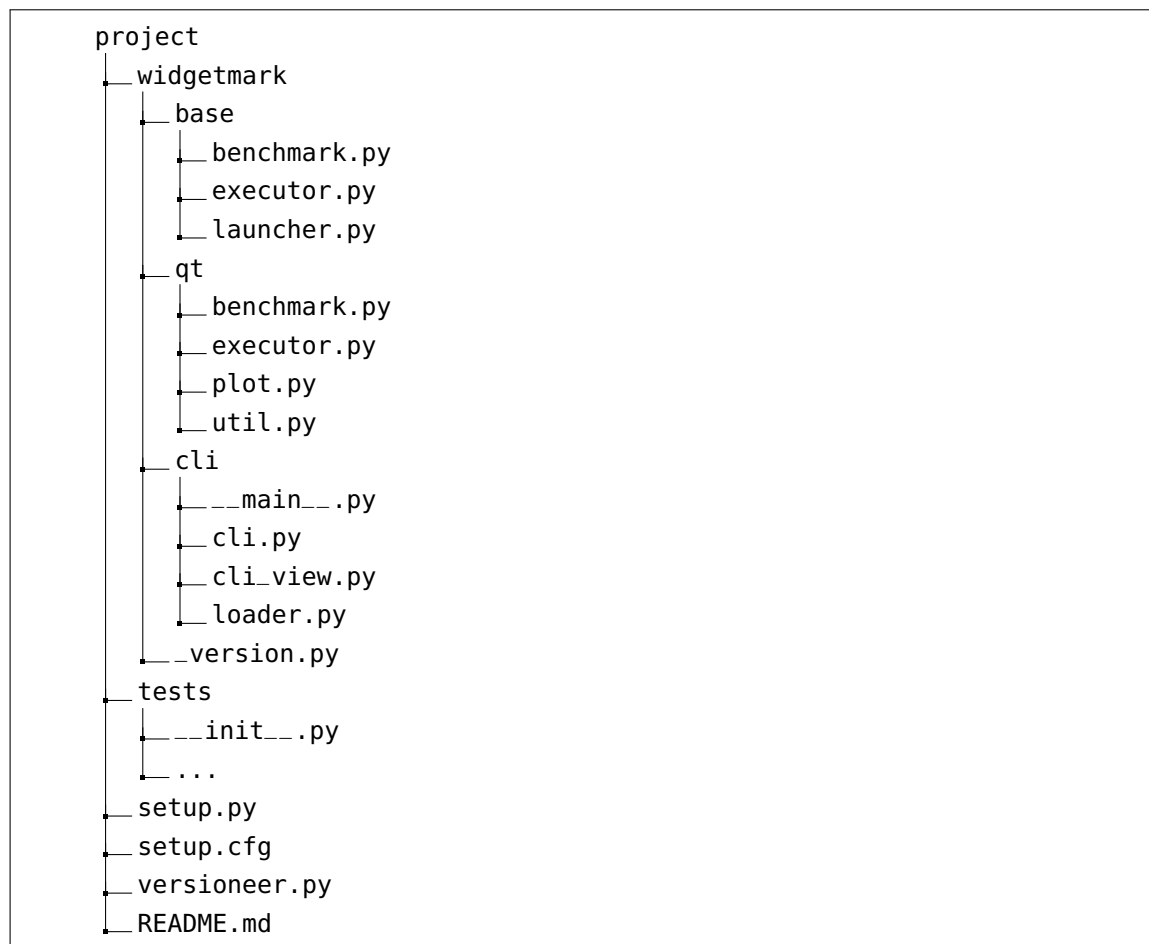


Abbildung 4.7.: Widgetmark project structure

4.4.2. 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 skip qt specific implementations of functionality in this package and only offer common functionality not related to GUI operations. Qt specific implementations will be part of the `widgetmark.qt` package.

benchmark.py

The first class for our implementation is the use case interface. Python does not support interfaces, but offers abstract classes we can use to define our interface by simply not provide any implementations for functions.

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

When distinguish three different types of information that can be defined in a use case: functions all subclasses have to implement, attributes subclasses have to implement and attributes subclasses can implement. 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 class attributes.

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

For implementing mandatory 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 actually depict a mandatory attribute on a class level, the abstract property can be implemented as a class attribute in subclasses.

```
1     @property
2     @abstractmethod
3     def goal(self) -> float:
4         pass
```

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

With this implementation, there is no visual difference between defining mandatory or optional attributes in use case classes, but we can enforce, that all mandatory attributes have been implemented. Both, class attributes and abstract properties can also be accessed the same way after the class is initialized using `instance.attribute`.

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 information we have recorded

from the use case. 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 we offer convenient properties for checks on the results. An example is a `failed` property that can be used to check for exceptions instead of comparing the `result.exception is None` property against `result.failed` .

```
1 class UseCaseResult:
2
3     def __init__(self,
4                 use_case: Union["UseCase", str],
5                 operations_per_second: Optional[float] = None,
6                 profiling_stats: Optional[pstats.Stats] = None,
7                 timed_out: bool = False,
8                 exception: Union[Exception, None] = None):
9         self._use_case = use_case
10        self._ops_ps = operations_per_second
11        self._timed_out = timed_out
12        self._exception = exception
13        self._profiling_stats = profiling_stats
14    @property
15    def operations_per_second(self):
16        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 to keep a direct reference to the use case. Since our goal is to keep the base implementation free from GUI framework specific functionality, we won't subclass any Qt window classes like `QtWidgets.QMainWindow` .

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 4.4 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 control 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. As for the benchmarking window, this implementation will be done in a fitting executor subclass in package `widgetmark.qt` .

For all GUI independent functionalities, this class does provide implementations. As seen in 4.4 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.

```

1  def launch(self, profile: bool = False) -> UseCaseResult:
2      """Start the execution of the Benchmark"""
3      if self._window is None:
4          raise Exception("Before launching the Executor a benchmarking "
5                          "window has to be set.")
6      self._window.use_case.runtime_context = self._runtime_context
7      self._last_time = time.time()
8      try:
9          self._window.integrate_use_case_widget()
10     except Exception as e:
11         # Catch exceptions raised when executing the Use Case
12         logger.debug(traceback.format_exc())
13         return UseCaseResult(
14             use_case=self._window.use_case,
15             exception=e,
16         )
17     self._window.prepare_window()
18     self._launch()
19     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 operation `_redraw_operation`, which is responsible for calling the operation on the window, collect timing information and control the profiler.

```

1  def _redraw_operation(self):
2      """
3      Execute one operation which triggers the widget redraw.
4      This function should be used in the implementation of
5      _launch().
6      """
7      try:
8          self._profile()
9          self._window.operate()
10     except Exception as e:
11         # Catch exceptions raised when executing the Use Case
12         self._result = self._complete(exception=e)
13         logger.debug(traceback.format_exc())
14     self._record_current_time()
15     self._check_if_completed()

```

For recording the current time-stamp we can use the `time()` function of the python package `Pythontime`, which returns us the current time as floating point number. From this time-stamp and the last recorded one we can calculate the delta timing for the last step. The current time will be saved in an instance attribute for the next operate.

```

1  def _record_current_time(self):
2      """Save the current timestamp for later evaluation."""
3      now = time.time()
4      delta = now - self._last_time
5      self._last_time = now

```

```
6 self._timing_recorder.append(delta)
```

The protected function `_profile` is responsible for controlling the profiler, which is used to profile the use case operation and combine the profiling stats from each step to a single profiling stat. Before the old step the current Profile will be disabled, the stats are added to the previously saved ones and the new profile will be enabled.

```
1 def _profile(self) -> None:
2     """
3     Add the last recorded profile to the profiling stats and restart the
4     profiler, if the use case has the profile flag set.
5     """
6     if self.use_case.profile:
7         if self._profile_stats is None:
8             self._profile_stats = pstats.Stats()
9         if self._current_profiler is not None:
10            self._current_profiler.disable()
11            self._profile_stats.add(self._current_profiler)
12            # TODO: use clear() instead of always creating a new profile
13            self._current_profiler = cProfile.Profile()
14            self._current_profiler.enable()
```

The last important function is `_check_if_completed()` function, 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.

```
1 def _check_if_completed(self):
2     """
3     Check if the Benchmark is completed. If the benchmark is completed,
4     call the completion handler, which is implemented by the executor
5     subclass.
6     """
7     if self.completed:
8         self._result = self._complete()
9     elif self.timed_out:
10        logger.debug(f"Use case {type(self.use_case).__name__} "
11                    f"timed out after taking more than "
12                    f"{self.use_case.timeout} seconds.")
13        self._result = self._complete(timed_out=True)
14        self._execution_counter += 1
```

launcher.py

The module `launcher` contains the class `Launcher`. As figure 4.3 shows, the purpose of the Launcher class is to create fitting benchmarking windows and executors for the use cases it was passed on initialization. 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 fitting classes are imported using

the `importlib` module which python offers. To make sure we import only classes that we want, we filter the found classes by their name and their type. The filtering by name is for cases the user specifies a specific name or name pattern on the command line, the type filtering is for not accidentally trying to initialize classes which do not define use cases. This way use case files can safely contain other class definitions without leading to errors.

```

1  @staticmethod
2  def _import(file_name: str,
3             class_type: Type,
4             name_filter: str = "") -> List[Type[UseCase]]:
5      module_name: str = os.path.basename(file_name)
6      logger.debug(f"Search for benchmark in module {module_name}.")
7      path: str = file_name
8      spec = importlib.util.spec_from_file_location(module_name, path)
9      benchmark_module = importlib.util.module_from_spec(spec)
10     if (spec.loader is not None
11         and isinstance(spec.loader, importlib.abc.Loader)):
12         spec.loader.exec_module(benchmark_module)
13     else:
14         raise InvalidBenchmarkError("Module {module_name} could not "
15                                    "be loaded.")
16     use_case_classes = Launcher._get_types_from_module(benchmark_module,
17                                                         class_type)
18     if name_filter:
19         logger.debug(f"Filter the found {len(use_case_classes)} "
20                    f"use case classes for name {name_filter}")
21         use_case_classes = [cls for cls in use_case_classes
22                             if re.match(name_filter, cls.__name__)]
23     if not use_case_classes:
24         raise InvalidBenchmarkError(f"No Use Case class could be "
25                                    f"found in module {module_name}.")
26     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 parameter 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 initialized use case instance. If the use case class defines for example the parameters `'number': [0, 1]`, we can access the value of this parameter in the use case class during execution by calling `self.number`.

```

1  def run(self, profile: bool = False) -> List[UseCaseResult]:
2      """Run all loaded use cases."""
3      results: List[UseCaseResult] = []
4      for use_case_class in self._use_case_classes:
5          combs = self.get_params_combination(use_case_class.parameters)
6          for comb in combs:
7              try:
8                  use_case: UseCase = use_case_class()

```

```

9         if profile:
10             use_case.profile = True
11             params_list = []
12             for name, value in comb.items():
13                 setattr(use_case, name, value)
14                 params_list.append(Launcher._readable_string(value))
15             use_case.params = ", ".join(params_list)
16         except TypeError:
17             results.append(
18                 UseCaseResult(
19                     use_case=use_case_class.__name__,
20                     exception=InvalidUseCaseDefinitionError(),
21                 ),
22             )
23             logger.debug(traceback.format_exc())
24             continue
25         executor = BackendResolver.executor_type(use_case.backend)()
26         window = BackendResolver.window_type(
27             use_case.backend)(use_case=use_case)
28         executor.set_window(window=window)
29         results.append(executor.launch(profile=profile))
30     return results

```

To resolve the right window and executor implementation for the backend defined by the use case, we have the class `BackendResolver` which does iterates through a classes subclass using the underscore function `__subclasses__()` to find one fitting to the requested backend. For this each subclass defines a class attribute `gui_backend`, which can be compared to the `backend` class attribute of the use case classes.

4.4.3. Subpackage widgetmark.qt

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

benchmark.py

This module provides a Qt based implementation of the `AbstractBenchmarkingWindow` based on `QtWidgets.QMainWindow`. Since Python supports multi inheritance, the Qt benchmarking window can be implemented as follows. The `make_qt_abc_meta()` function in this class definition is for solving the meta class conflict between qt classes and the `ABCMeta` from the `abc` package. This is achieved by returning a metaclass that is derived from both the meta class of `QMainWindow` and `ABCMeta`. Its implementation can be found in the `util.py` module of the same package.

```

1 class QtWindow(AbstractBenchmarkWindow,
2               QMainWindow,
3               metaclass=make_qt_abc_meta(QMainWindow)): # type: ignore

```


executor.py

This module provides a Qt based implementation of the `AbstractBenchmarkExecutor` based on `QtCore.QObject`. The executor has to start a Qt application and implement two functions based on it: `_launch()` and `_complete()`. Initialization of the benchmark execution loop is the purpose of `_launch`. To make sure, that no pending events are skipped, we want to give back control to Qt's event loop after each time our operation we want to benchmark is executed. In Qt we can achieve such a loop by using the `QtCore.QTimer` with a timeout of zero seconds. Such a timer will timeout everytime it is possible. 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.

```

1  def _launch(self):
2      self._timer: QTimer = QTimer()
3      self._timer.timeout.connect(self._redraw_operation)
4      self._timer.start(0)
5      self._app.exec()

```

After the executor's base notices in its `_check_if_completed` function, that either the timeout or the maximum repeat counter is reached, it will call the `_complete` function. Its purpose is not only to wrap the recorded performance figures in a `UseCaseResult` instance, but also to stop the execution loop. Since the loop is based on a `QtCore.QTimer`, we can use its `stop()` function to achieve this.

```

1  def _complete(self,
2      timed_out: bool = False,
3      exception: Optional[Exception] = None) -> UseCaseResult:
4      self._timer.stop()
5      self._app.closeAllWindows()
6      self._app.quit()
7      self._app.processEvents()
8      return UseCaseResult(use_case=self.use_case,
9                          operations_per_second=self.operations_per_second,
10                         profiling_stats=self._profile_stats,
11                         timed_out=timed_out,
12                         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.

```

1  @staticmethod
2  def _prepare_qapp():
3      """Prepare QApplication instance and install event filters."""
4      app = QApplication.instance()
5      if app is None:
6          app = QApplication([])
7          app.installEventFilter(

```

```
8         InteractionEventFilter.get_event_filter(app))
9     return app
```

Since the window and the widget is exposed during the benchmark execution, the user is able to interact with it. Since these interactions are not part of the use case, they would affect the results negatively, since their work effort would be counted as work produced by the use case. Because of this, we have to filter out all operations, which are not directly caused by operations of the use case. As described in section 2.3.3, Qt offers us the possibility to install event filters, which can filter out all events coming from the window system. For our purposes we want to filter out all mouse interactions using the following class derived from `QtCore.QObject`.

```
1 class InteractionEventFilter(QObject):
2
3     filter = [QEvent.MouseButtonDblClick,
4               QEvent.MouseButtonPress,
5               QEvent.MouseButtonRelease,
6               QEvent.MouseTrackingChange,
7               QEvent.GrabMouse,
8               QEvent.UngrabMouse,
9               QEvent.Move,
10              QEvent.DragEnter,
11              QEvent.DragLeave,
12              QEvent.DragMove]
13
14     def eventFilter(self, o: QObject, e: QEvent):
15         if e.type() in InteractionEventFilter.filter:
16             e.ignore()
17             return True
18         return super().eventFilter(o, e)
```

plot.py

The module `widgetmark.qt.plot` contains the implementation of the abstraction layer for plotting operations. As described in section 4.3.2 and in figure 4.6, we want to introduce an abstraction layer for the plotting libraries we are focusing on to write benchmark use cases that are reusable for multiple plotting libraries. 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 which returns the fitting subclass to the passed plotting library.

```
1 class AbstractBasePlot(
2     QtWidgets.QWidget,
3     metaclass=make_qt_abc_meta(QtWidgets.QWidget), # type: ignore
4 ):
5
6     library: PlottingLibraryEnum = None # type: ignore
7     item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {}
8
```

```

9     @classmethod
10     def using(cls: Type["AbstractBasePlot"],
11               library: PlottingLibraryEnum) -> "AbstractBasePlot":
12         return cls.get_subclass_for_lib(library=library)()
13
14     @classmethod
15     def get_subclass_for_lib(
16         cls: Type["AbstractBasePlot"],
17         library: Union[PlottingLibraryEnum, str],
18     ) -> Type["AbstractBasePlot"]:
19         fitting_classes: List[Type[AbstractBasePlot]] = [
20             c for c in AbstractBasePlot.all_subclasses(cls)
21             if library == c.library and c.library is not None
22         ]
23         if not fitting_classes:
24             raise ValueError(f"No fitting subclass was found for the "
25                               f"plot library {library}.")
26         elif len(fitting_classes) > 1:
27             logger.debug(f"Multiple subclasses were found for the "
28                           f"plot library {library}.")
29         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 displays some data.

```

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

```

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

```

1 class PyQtGraphPlot(pg.PlotWidget, AbstractBasePlot):
2
3     library = PlottingLibraryEnum.PYQTGRAPH
4
5     item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {
6         DataItemType.CURVE: (pg.PlotDataItem, {"pen": "w", "symbol": None}),
7         DataItemType.SCATTER: (pg.PlotDataItem, {"pen": None, "symbol": "o"}),
8     }
9     def add_item(self, item_type) -> "PyQtGraphDataItem":
10         item_class, params = self.item_type_mapping.get(item_type, (None, {}))
11         if item_class is not None and isinstance(params, Dict):
12             item = item_class(**params)
13             self.addItem(item)
14             return PyQtGraphDataItem(item)
15         else:
16             raise ValueError(f"For data item type {item_type} is no fitting "
17                               f"item is defined.")

```

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

```
1 class MatPlotLibPlot(AbstractBasePlot):
2
3     library = PlottingLibraryEnum.MATPLOTLIB
4
5     item_type_mapping: Dict[DataItemType, Tuple[Type, Dict]] = {
6         DataItemType.CURVE: (Line2D, {
7             "xdata": [],
8             "ydata": [],
9             "linestyle": "-",
10        }),
11        DataItemType.SCATTER: (Line2D, {
12            "xdata": [],
13            "ydata": [],
14            "linestyle": "",
15            "marker": "o",
16        }),
17    }
18
19    def add_item(self, item_type) -> "MatPlotLibDataItem":
20        item_class, params = self.item_type_mapping.get(item_type, (None, {}))
21        if item_class is not None and isinstance(params, Dict):
22            item = item_class(**params)
23            self._plot.add_artist(item)
24            return MatPlotLibDataItem(self._canvas, item)
25        else:
26            raise ValueError(f"For data item type {item_type} is no fitting "
27                             f"item is defined.")
```

Both functions will return objects derived from the same `AbstractDataItem`, which allows reading, writing and adding data in form of a two dimensional numpy array containing x and y-values, which fits our planned usage in listing 4.3. In our use cases, we can add the plotting library we want to benchmark as a parameter in its `parameters` section which we can then pass to the factory function `using()` of the `AbstractBasePlot` class. This way we can benchmark the exact same use case in two completely different plotting libraries without having to define two separate use cases.

4.4.4. Subpackage widgetmark.cli

The last package of our project's source code contains all modules related to the command line interface. Its purpose is accept user input to start the launcher with the files or use cases the user want as well as to present the results gathered from execution. Additionally it write the recorded profiles to separate files to a location defined by the user and start the visualization for them using the `snakeviz` python application. For defining command line arguments, the python package `argsparse` is used, which allows convenient parsing of arguments passed on the command line when executing the CLI module from the

command line. Additionally, a help option will be added, which explains the usage when the user executes `widgetmark --help`.

When installing `widgetmark` using `setuptools`, we can register wrap register our CLI as an console script. This allows us to make a Python function accessible from the command line. In the module main, we have our central main method, which initializes the CLI and executes it.

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

In the module `setup.py` we can register this function as a `console_script` follows and define the name of the produces executable. In our case this will simply be `'widgetmark'`

```
1 setup(
2     name="widgetmark",
3     entry_points={
4         "console_scripts": ["widgetmark = widgetmark.cli:main"],
5     },
6 )
```

4.5. Use Case Implementation

...

5. Performance optimization

5.1. Optimizing line graph performance

5.1.1. GPU accelerated Rendering

...

5.2. Optimizing scatter plot performance

5.2.1. Incremental data updates

...

6. Evaluation

...

6.1. Benchmark changes to Line Graph

...

6.2. Benchmark changes to Scatter Plot

...

7. Conclusion

7.1. Summary

...

7.2. Outlook

...

Literatur

- [1] Ajitsaria Abhinav. *What is the Python Global Interpreter Lock (GIL)?* Juni 2018. URL: <https://realpython.com/python-gil/>.
- [2] “Benchmarking als Instrument eines wettbewerbsorientierten Performance Management”. In: *Marketing Performance*. Gabler, S. 237–249. DOI: 10.1007/978-3-8349-0664-9_11. URL: https://doi.org/10.1007/978-3-8349-0664-9_11.
- [3] Jasmin Blanchette. *Another Look at Events*. 2004. URL: <https://doc.qt.io/archives/qq/qq11-events.html#eventhandlingandfiltering>.
- [4] Luke Campagnola. *PyQtGraph Documentation*. 2011. URL: <http://www.pyqtgraph.org/documentation/index.html>.
- [5] CERN. *A day in the CERN Control Center*. URL: <https://home.cern/news/news/accelerators/day-cern-control-centre>.
- [6] 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>.
- [7] CERN. *Beams Department*. URL: <https://beams.web.cern.ch>.
- [8] CERN. *Home*. URL: <https://home.cern>.
- [9] CERN. *Lhc Report: The final days of Run 2*. URL: <https://home.cern/news/news/accelerators/lhc-report-final-days-run-2>.
- [10] CERN. *Linear Accelerator 4*. URL: <https://home.cern/science/accelerators/linear-accelerator-4>.
- [11] CERN. *Our history*. URL: <https://home.cern/about/who-we-are/our-history>.
- [12] CERN. *Processing: What to record?* URL: <https://home.cern/science/computing/processing-what-record>.
- [13] CERN. *The Large Hadron Collider*. URL: <https://home.cern/science/accelerators/large-hadron-collider>.
- [14] CERN. *The Low Energy Ion Ring*. URL: <https://home.cern/science/accelerators/low-energy-ion-ring>.
- [15] CERN. *Who we are*. URL: <https://home.cern/about/who-we-are>.
- [16] Stéphane Deghayes und Eve Fortescue-Beck. *Introduction to the BE-CO Control System*. 2019 Edition. CERN, 2019.
- [17] Ben Dotson. *How Particle Accelerators Work*. 2014. URL: <https://www.energy.gov/articles/how-particle-accelerators-work>.

- [18] ATLAS Experiment. *Live Collisions in the ATLAS Detector*. URL: <https://atlas-public.web.cern.ch/atlas-live>.
- [19] ATLAS Experiment. *Trigger and Data Acquisition System*. URL: <https://atlas-public.web.cern.ch/discover/detector/trigger-daq>.
- [20] Stephen Few. “Data Visualization - Past, Present, and Future”. In: (2007). URL: https://www.perceptualedge.com/articles/Whitepapers/Data_Visualization.pdf.
- [21] Michael Firendly. *Milestones in the History of Data Visualization*. Nov. 2003. URL: <http://datavis.ca/papers/carme2003-2x2.pdf>.
- [22] Python Software Foundation. *Extending Python with C or C++*. 2019. URL: <https://docs.python.org/3/extending/extending.html>.
- [23] 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>.
- [24] Michael Friendly. “Milestones in the History of Data Visualization: A Case Study in Statistical Historiography”. In: *Classification: The Ubiquitous Challenge*. Hrsg. von C. Weihs und W. Gaul. New York: Springer, 2005, S. 34–52. URL: <http://www.math.yorku.ca/SCS/Papers/gfkl.pdf>.
- [25] Robert B. Haber und David A. McNabb. “Visualization Idioms: A Conceptual Model for Scientific Visualization Systems”. In: ().
- [26] John D. Hunter. *History*. 2008. URL: <https://matplotlib.org/3.1.1/users/history.html>.
- [27] G. Kruk und M. Peryt. *JDATAVIEWER – JAVA-BASED CHARTING LIBRARY*. CERN, Geneva, Switzerland, Nov. 2009.
- [28] Riverbank Computing Limited. *What is PyQt?* 2018. URL: <https://www.riverbankcomputing.com/software/pyqt/intro>.
- [29] Riverbank Computing Limited. *What is SIP?* 2018. URL: <https://www.riverbankcomputing.com/software/sip/intro>.
- [30] The Qt Company Ltd. *Graphics View Framework | Qt Widgets 5.14.0*. 2019. URL: <https://doc.qt.io/qt-5/graphicsview.html>.
- [31] 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>.
- [32] The Qt Company Ltd. *Signals and Slots | Qt Core 5.14.0*. 2019. URL: <https://doc.qt.io/qt-5/signalsandslots.html>.
- [33] The Qt Company Ltd. *The Event System | Qt 4.8*. 2016. URL: <https://doc.qt.io/archives/qt-4.8/eventsandfilters.html>.
- [34] Igor Milovanovic. *Python data visualization cookbook*. Birmingham: Packt Publ., 2013. URL: <https://cds.cern.ch/record/1641733>.
- [35] Alan D. Moore. *Mastering GUI programming with Python: develop impressive cross-platform GUI applications with PyQt*. Packt Publishing, Limited, 2019.

-
- [36] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).
 - [37] Alessandro Pasotti. *Python SIP C++ bindings tutorial*. Sep. 2014. URL: <https://www.itopen.it/python-sip-c-bindings-experiments/>.
 - [38] Maciej Peryt und 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.
 - [39] Selan dos Santos und Ken Brodlie. “Gaining understanding of multivariate and multidimensional data through visualization”. In: *Computers & Graphics* 28.3 (2004), S. 311–325. DOI: 10.1016/j.cag.2004.03.013.
 - [40] Quincy Smith. *The Best Python Data Visualization Libraries*. Nov. 2019. URL: <https://www.fusioncharts.com/blog/best-python-data-visualization-libraries/>.
 - [41] *Standard Performance Evaluation Corporation*. URL: <http://www.spec.org/>.
 - [42] The Matplotlib development team. *Matplotlib*. 2019. URL: <https://matplotlib.org/3.1.1/index.html>.
 - [43] The Matplotlib development team. *Usage Guide*. 2019. URL: <https://matplotlib.org/3.1.1/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py>.
 - [44] Trolltech. *Events and Event Filters*. 2005. URL: <https://doc.qt.io/archives/3.3/eventsandfilters.html>.
 - [45] Reinhold P. Weicker. “An overview of common benchmarks”. In: *Computer* 23.12 (1990), S. 65–75. DOI: 10.1109/2.62094.
 - [46] *Which programming language is fastest?* URL: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>.

A. Anhang

A.1. PyQt

```
1 import sys
2 from qtpy import QtWidgets
3
4
5 class MainWindow(QtWidgets.QMainWindow):
6
7     def __init__(self, *args):
8         super().__init__(*args)
9         self.setup_widgets()
10        self.setup_layout()
11        self.set_widgets_text()
12        self.show()
13
14    def setup_widgets(self):
15        # Upper part of the window (horizontally aligned widgets)
16        self.central_widget = QtWidgets.QWidget(self)
17        self.central_layout = QtWidgets.QVBoxLayout(self.central_widget)
18        self.upper_layout = QtWidgets.QHBoxLayout(self.central_widget)
19        self.check_box = QtWidgets.QCheckBox(self.central_widget)
20        self.label = QtWidgets.QLabel(self.central_widget)
21        self.push_button = QtWidgets.QPushButton(self.central_widget)
22        # Lower part of the window (tabs)
23        self.tab_widget = QtWidgets.QTabWidget(self.central_widget)
24        self.content_tab_1 = QtWidgets.QWidget(self.tab_widget)
25        self.content_tab_2 = QtWidgets.QWidget(self.tab_widget)
26        self.layout_tab_1 = QtWidgets.QGridLayout(self.content_tab_1)
27        self.layout_tab_2 = QtWidgets.QGridLayout(self.content_tab_2)
28        self.label_tab_1 = QtWidgets.QLabel(self.content_tab_1)
29        self.label_tab_2 = QtWidgets.QLabel(self.content_tab_2)
30
31    def setup_layout(self):
32        # Set size and central widget
33        self.resize(360, 200)
34        self.setCentralWidget(self.central_widget)
35        # Fill upper half of the window
36        self.central_layout.addLayout(self.upper_layout)
37        self.upper_layout.addWidget(self.check_box)
38        self.upper_layout.addWidget(self.label)
39        self.upper_layout.addWidget(self.push_button)
```

```
40     self.central_layout.addWidget(self.tab_widget)
41     # Fill lower half of the window
42     self.tab_widget.addTab(self.content_tab_1, "")
43     self.tab_widget.addTab(self.content_tab_2, "")
44     self.layout_tab_1.addWidget(self.label_tab_1)
45     self.layout_tab_2.addWidget(self.label_tab_2)
46     self.tab_widget.setCurrentIndex(0)
47
48     def set_widgets_text(self):
49         self.setWindowTitle("Example Application")
50         # Set texts in the upper half of the window
51         self.check_box.setText("CheckBox")
52         self.label.setText("TextLabel")
53         self.push_button.setText("PushButton")
54         # Set texts in the lower half of the window
55         self.tab_widget.setTabText(0, "Tab 1")
56         self.tab_widget.setTabText(1, "Tab 2")
57         self.label_tab_1.setText("This is the content of Tab I")
58         self.label_tab_2.setText("This is the content of Tab II")
59
60
61 if __name__ == "__main__":
62     app = QtWidgets.QApplication(sys.argv)
63     _ = MainWindow()
64     sys.exit(app.exec())
```

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

```
1 import sys
2 from qtpy import QtWidgets, QtGui, QtCore
3
4
5 class ChangingLabel(QtWidgets.QLabel):
6
7     def mousePressEvent(self, e: QtGui.QMouseEvent):
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
15
16 class LabelMouseClickedFilter(QtCore.QObject):
17
18     def eventFilter(self, o: QtCore.QObject, e: QtCore.QEvent):
19         """Filter out all Mouse Clicks, single or double"""
20         intercept = [QtCore.QEvent.MouseButtonDblClick,
21                     QtCore.QEvent.MouseButtonPress]
22         if e.type() in intercept and isinstance(o, QtWidgets.QLabel):
```

```

23         return True
24         return super().eventFilter(o, e)
25
26
27 class MainWindow(QtWidgets.QMainWindow):
28
29     def __init__(self, app: QtWidgets.QApplication):
30         super().__init__()
31         self._app = app
32         self.setWindowTitle("Event Demo")
33         cw = QtWidgets.QWidget()
34         self.setCentralWidget(cw)
35         cl = QtWidgets.QGridLayout()
36         cw.setLayout(cl)
37         self._filter = LabelMouseClickFilter(parent=app)
38         self._label_1 = ChangingLabel("I react to Mouse Clicks!")
39         self._label_2 = ChangingLabel("I react to Mouse Clicks as well!")
40         self._checkbox = QtWidgets.QCheckBox("Install Event Filter")
41         self._button = QtWidgets.QPushButton("Send Left Click to Labels")
42         self._checkbox.stateChanged.connect(self._install_filter)
43         self._button.pressed.connect(self._send)
44         cl.addWidget(self._label_1, 0, 0)
45         cl.addWidget(self._label_2, 0, 1)
46         cl.addWidget(self._checkbox, 1, 0)
47         cl.addWidget(self._button, 1, 1)
48         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 an example window demonstrating qts event system

A.2. PyQtGraph

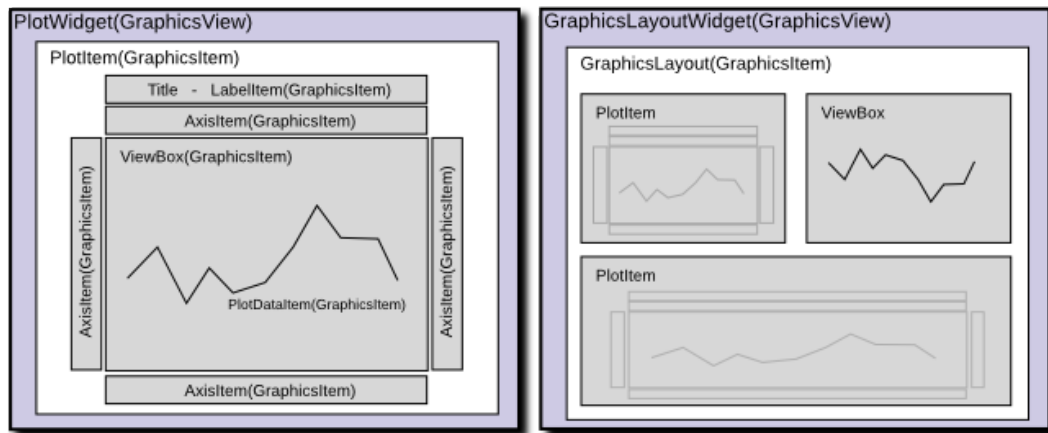


Abbildung A.1.: Different items involved in a PlotWidget.

Quelle: <http://www.pyqtgraph.org/documentation/plotting.html>

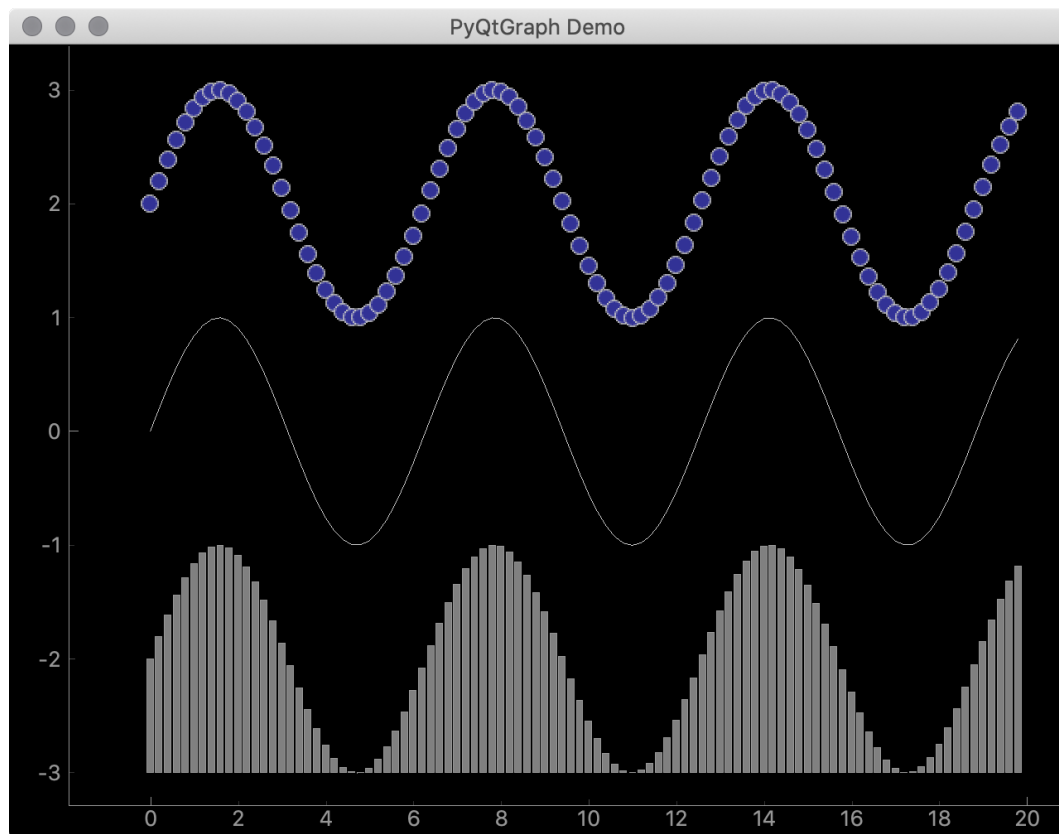


Abbildung A.2.: Window containing a plot created with PyQtGraph.

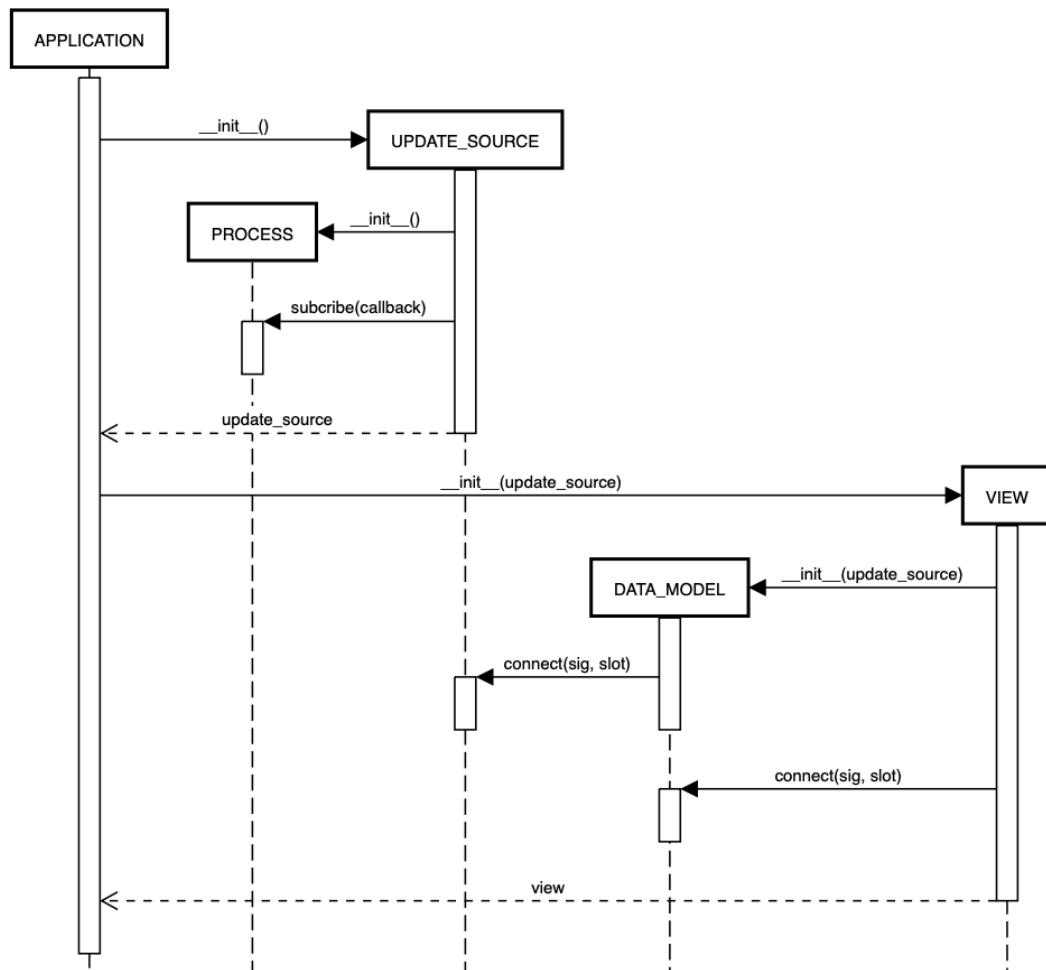


Abbildung A.3.: Initialization of the Live Plotting Architecture for PyQtGraph.

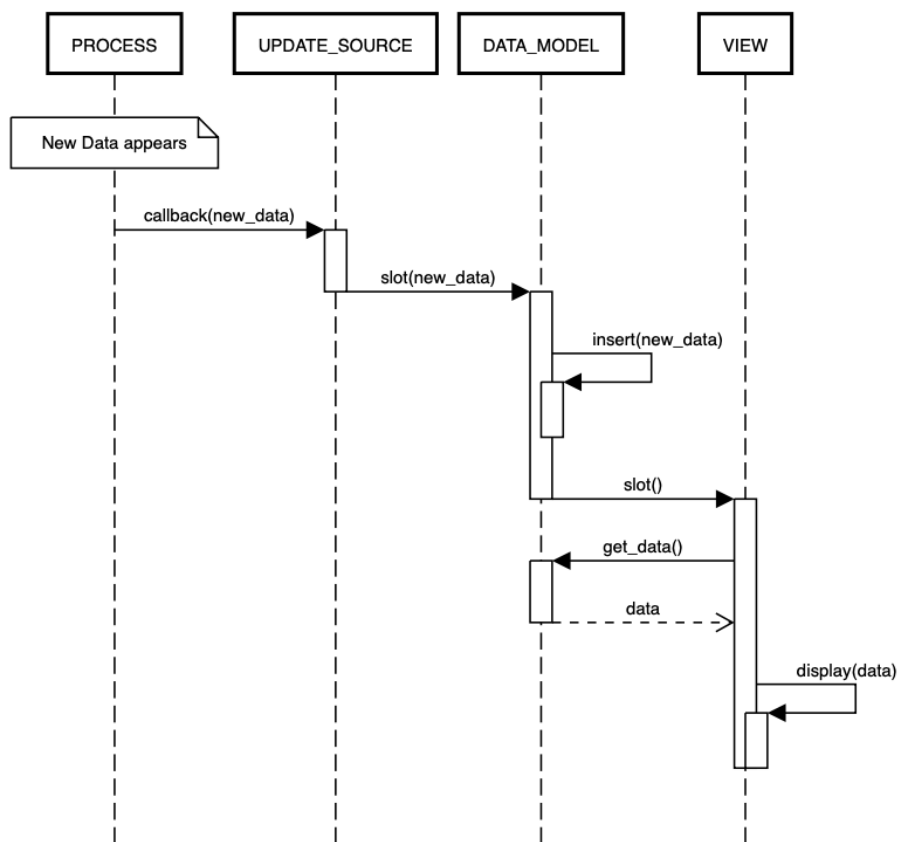


Abbildung A.4.: Update Propagation in the Live Plotting Architecture for PyQtGraph.

A.3. Matplotlib

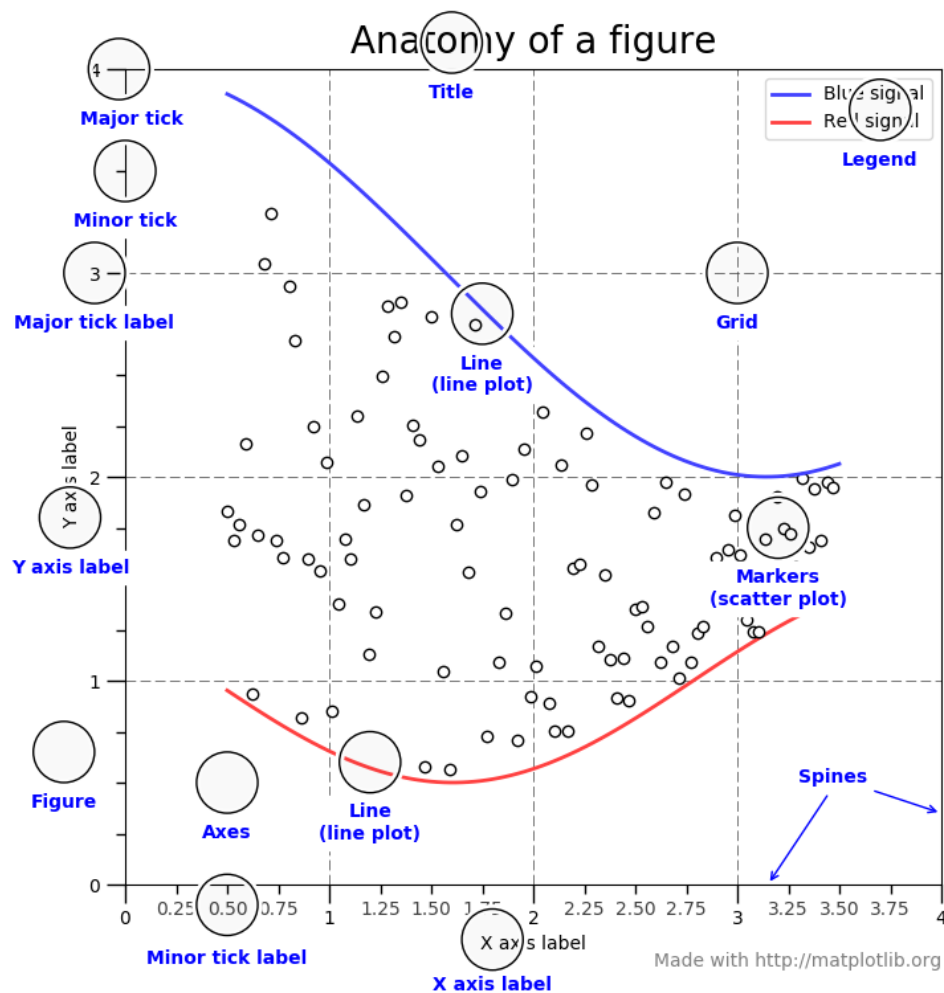


Abbildung A.5.: Different items involved in a Figure.

Quelle: <https://matplotlib.org/3.1.1/tutorials/introductory/usage.html>

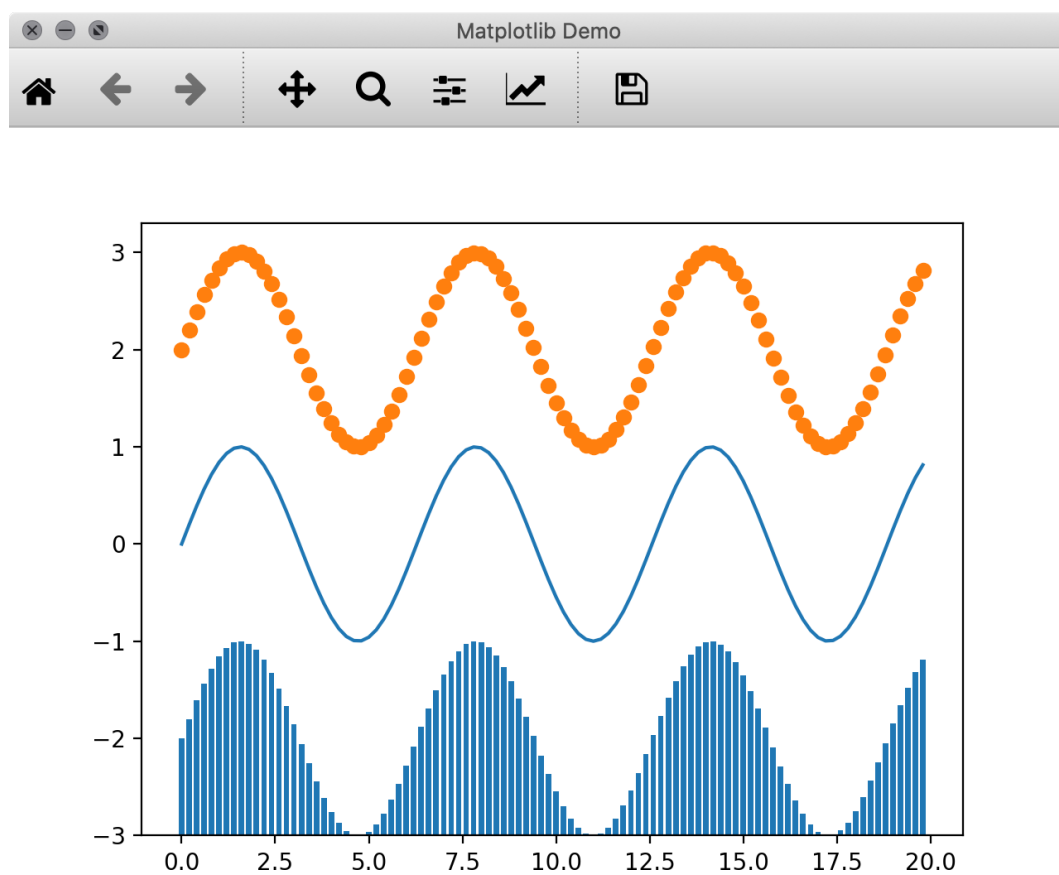


Abbildung A.6.: Window containing plot created with Matplotlib.