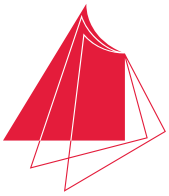

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

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

Abbildungsverzeichnis

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 orgranisation 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 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 devided 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 collission. 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 occuring 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 perfromance 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 measurment 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 thechnical 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 adaptions 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 earlies 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 utlized 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 carthesian 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 disaplying 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 then 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 alrady 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 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.

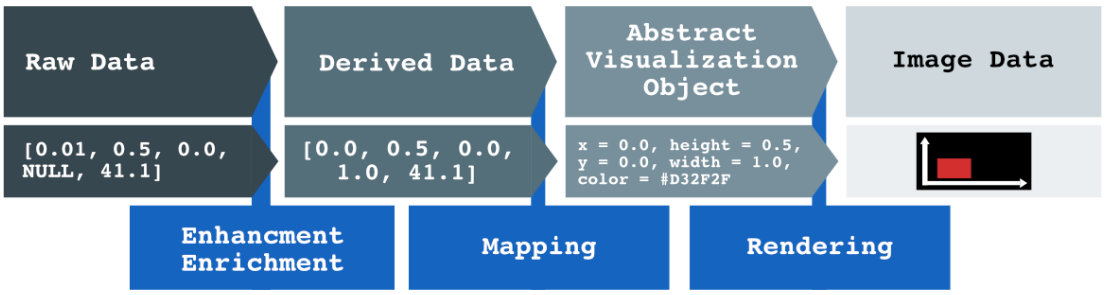

Abbildung 2.2.: Transformations and Resulting Data of the Data Visualization Pipeline

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 noice 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 graphcis, 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 acquistion 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 sofware sytems 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 Octobter 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 challange. 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 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]

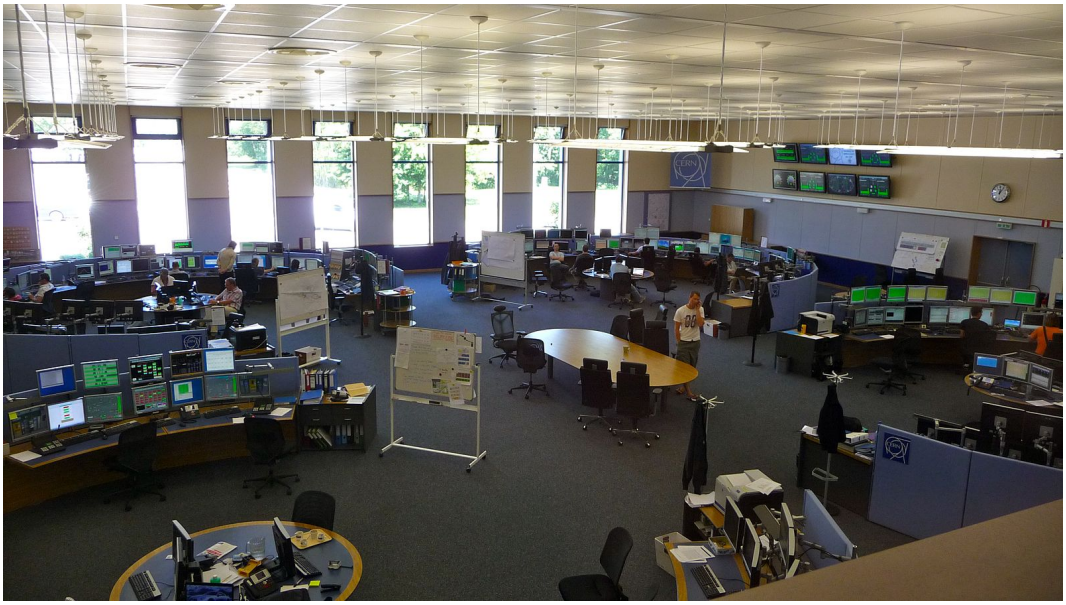

Abbildung 2.3.: CERN Control Center

Implementations of the second and third type of data visualization software systems can be found as well at CERN, especially for monitoring continously 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 runnning 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 categroized 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 accellerator 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 LHC. Its Vistar contains a graph at the top displaying the injection of particles, the energy level and the different cyles. [14]

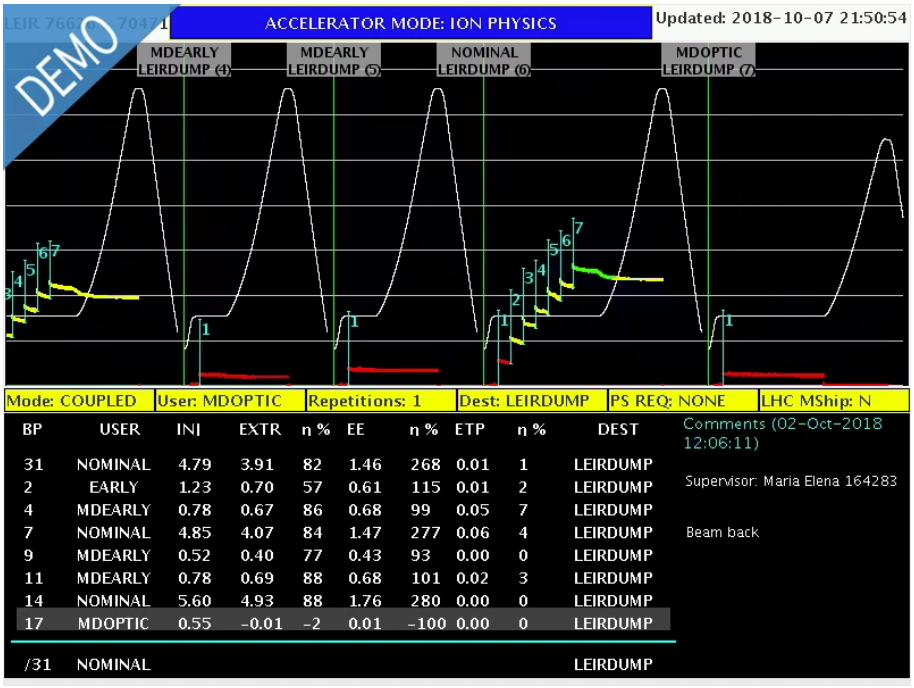

Abbildung 2.4.: LEIR Vistar in the CCC

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 develope 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 seperates 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 predeccessor 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, cofunder of Intel, one of the world's largest semiconductor chip manufacturer, the transistor count in a dense integrated cicuit 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 proccess more quickly through the given data. This appoach 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 transistorsn 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 efficienlty, 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 a 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 components 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 measurment to desribe 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 expensice integer operations. Until today, FLOPS can be found as a performance description for computing hardware, especially for GPUs. The GeForce RTX 2080, a to the time of writing very recent GPU from the GPU manufacturer NVIDIA, can theoretically reach up to 10.6 TeraFLOPS (TFLOPS) or $10.6 * 10^{12}$ FLOPS. [6]

The first ever program that was explicitly desgined to test the perfromance 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 repeatingly executed. By using different weights for different

operations, the results of the benchmark in the end could be described in a measurment 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 brough up into the world of benchmarking, was the consideration of energy efficiency next to performance as an interesting measurment. [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 apporach to benchmarking is Software Benchmarking.

2.2.2. Software Benchmarks

Next to benchmarking hardware, we can also measure the performance of software. Beanchmarking 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 archieved 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 langugage. [46]

2.2.3. Application Benchmarks

For most modern hardware, measurments 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 jourinalism about computer hardware you can find a very wide spread usage of application benchmarks to describe the performance of new realeased harware

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) Bechmarks 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 representative to the task we want to perform later on the hardware. Running rendering benchmarks on graphics hardware does maybe give us an 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 measureable. 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 basic principles of the underlying GUI framework first. This chapter will present an introduction into the core principles of the GUI Framework we will use for our Benchmarking Application.

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 genral introduction into Qt, which we will need for understanding the ideas behind our implementation.

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 devided into three main components:

QtCore offers non GUI related Core functionalities for applications.

QtGui offers GUI related data wrapper classes and utlity functions.

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

`QtWidgets.QApplication` is the central application class in Qt. This object will represent 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, we can use `QtWidgets.QMainWindow`. A clean way to create a window with 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.run()`, which will start the Main Event Loop of the Application.

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 use classes from the QtWidgets package.

```
import sys
from qtpy import QtWidgets


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hello World!")
        self.show()


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    _ = MainWindow()
    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 fill our window with content, in Qt we have to add widgets to it. The base class for every widget is `QtWidgets.QWidget`. Next to QWidget, the package QtWidgets offers many more widgets, which are ready to use and can be added to a window. All common UI building blocks, like labels, checkboxes, comboboxes and more can be found in this package. QWidget itself can also be used as a wrapper around multiple other widgets. Each widget will offer a parameter parent, which can be used to define a parent child hierarchy for all widgets starting with the main window of the application. 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 child widgets inside a parent widget. Next to the following three basic layouts, Qt also offers more complex ones with more functionality.

`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.

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 diffrent tabs. By utilizing these different widgets and laout options, you can build more complex window layouts as seen as seen visible in depiction 2.5. The source code for this window is appended and can be found in A.1. [35]

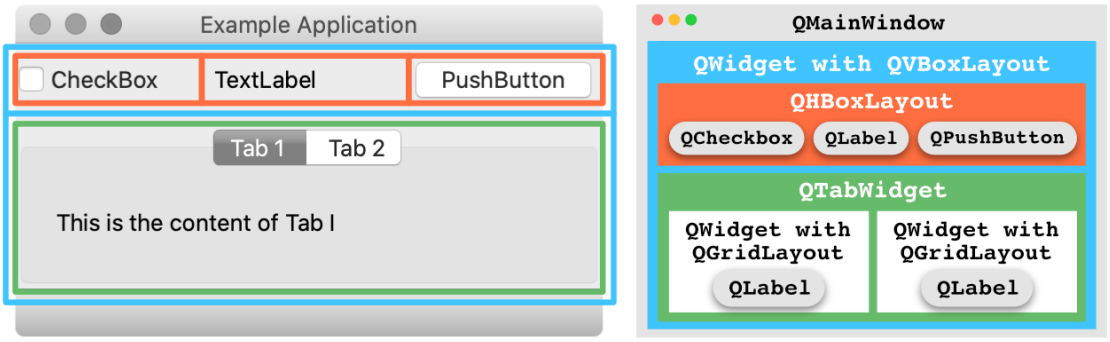

Abbildung 2.5.: Layouts and Widgets in a Qt Application Window

At this point, we have learned, how we can define the appereance 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, if for example a button is pressed. 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 `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 emits messages informing about changes in the state of an object. 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. Slots can also be called normally from code, since they are normal instance methods. 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 singals and slots.

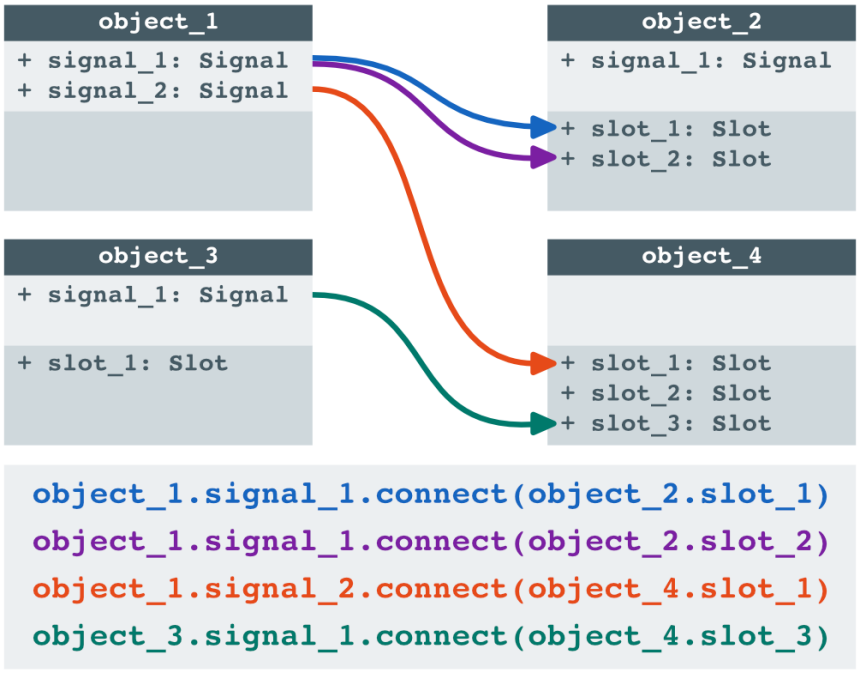

Abbildung 2.6.: Communication between objects through Signal and Slot Connections

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 perfromance losses coming through the usage of Signals and Slots are insignificant. [32, 35]

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 seperate Thread 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.

When working with multiples Threads in a Qt Application, it is important to know about the different connection types in Qt. When connecting Signals to Slots, Qt offers different Connection Types, which control, when and in which Thread the slot is executed. A `QtCore.Qt::DirectConnection` for example leads to the Slot being executed directly without returning to the main Event Loop (see 2.3.3) in the Thread of the emitting object, while a `QtCore.Qt::QueuedConnection` executes the Slot when returning to the event loop in the Thread where the receiving object lives. The default connection type `QtCore.Qt::AutoConnection` will decide between queued and direct connection depending on in which Thread the emitting and receiving object is living. [31]

2.3.3. The Event System

Qt Applications can very often be written without using parallel programming at all. Qt is an event-driven toolkit, which means, that applications are designed to wait for events and react to them appropriatly. 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 normally coming from the Window Manager while other events can also come either from the application code or the Qt Framework. Qt does not necessarely handle events immediately as they arrive, but queues them to work thorugh them sequentially at a later time. 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.QtCoreApplication.sendEvent()` or `QtCore.QtCoreApplication.postEvent()` .

The first will lead to immediate delivering and handling of the Event without any involvment of the Event Loop, while the later one will add the Event to the Event Queue, from where it will be deliverd by Qt when it is its turn. Listing 2.2 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.

```
from qtpy import QtCore, QtGui, QtWidgets

class ChangingLabelOne(QtWidgets.QLabel):
    pass

class MainWindow(QtWidgets.QMainWindow):

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

Listing 2.2: Positing a MousePressEvent 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 2.3 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.

```
from qtpy import QtWidgets, QtCore, QtGui

```

```
class ChangingLabelOne(QtWidgets.QLabel):

    def event(self, e: QtCore.QEvent) -> bool:
        """Change Label Font Size thorugh Mouse Click"""
        if e.type() == QtCore.QEvent.MouseButtonPress:
            font: QtGui.QFont = self.font()
            if e.button() == QtCore.Qt.RightButton:
                font.setPointSize(font.pointSize() - 1)
            elif e.button() == QtCore.Qt.LeftButton:
                font.setPointSize(font.pointSize() + 1)
            self.setFont(font)
            return True
        else:
            return super().event(e)
```

Listing 2.3: 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 2.4 shows the same Use Case as before but achieved through overwriting the specific event handler dedicated to Mouse Button Clicks.

```
from qtpy import QtWidgets, QtCore, QtGui

class ChangingLabelTwo(QtWidgets.QLabel):

    def mousePressEvent(self, e: QtGui.QMouseEvent):
        """Change Label Font Size thorugh Mouse Click"""
        font: QtGui.QFont = self.font()
        if e.button() == QtCore.Qt.RightButton:
            font.setPointSize(font.pointSize() - 1)
        elif e.button() == QtCore.Qt.LeftButton:
            font.setPointSize(font.pointSize() + 1)
        self.setFont(font)
```

Listing 2.4: 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 intalled on any `QtCore.QObject` and is working for it and its children. To install it globally for the whole application, it cann be installed on the `QWidgets.QApplication` instance as you can see as an example in listing 2.5. As before, we have to return `True` to prevent Qt from further propagating the event to other widgets.

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

class MouseClickFilter(QtCore.QObject):

```

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

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

Listing 2.5: 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 pronciples 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 containg 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 simliar 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 existance 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 repeatadly new widgets, for example when opening new dialogs in an application. To free the memory occupied by an widget's C++ object, we have to tell Qt, as seen in listing 2.6, to delte the object and its children.

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

Listing 2.6: 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 truely 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 parrallel 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 2.7 shows those two options.

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

Listing 2.7: 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 wil 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]

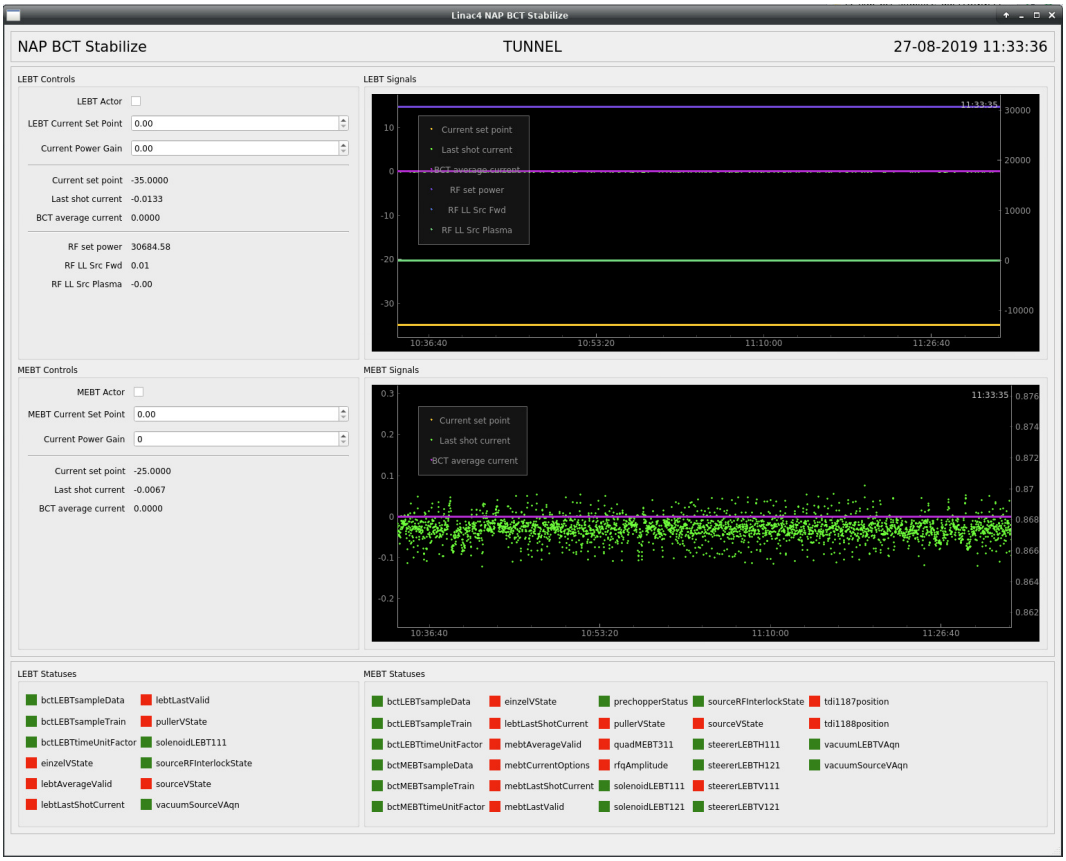

Abbildung 3.1.: Screenshot of the Linac4 Source Gui

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 containg 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 reffered 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 thourgh 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 devided 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.4.

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

class MatplotlibWindow(QtWidgets.QMainWindow):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Data
        x = np.array(range(100)) * 0.2
        y = np.sin(x)
        # Plot + Navigation Bar
        self._canvas = FigureCanvas(Figure())
        self._plot = self._canvas.figure.subplots()
        self._toolbar = NavigationToolbar(canvas=self._canvas, parent=self)
        # Data Visualization
        curve = self._plot.plot(x, y, "-")
        scatter = self._plot.plot(x, y + 2, "o")
        bars = self._plot.bar(x=x, height=y + 1, bottom=-3, width=0.15)
        # Setup Window Content
        layout = QtWidgets.QVBoxLayout()
        layout.setSpacing(0)
        layout.setContentsMargins(0, 0, 0, 0)
        layout.addWidget(self._toolbar)
        layout.addWidget(self._canvas)
        central_widget = QtWidgets.QWidget()
        central_widget.setLayout(layout)
        self.setCentralWidget(central_widget)
        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]

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

class PyQtGraphWindow(QtWidgets.QMainWindow):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Data
        x = np.array(range(100)) * 0.2
        y = np.sin(x)
        # Plot
        self._plot = pyqtgraph.PlotWidget()
        # Data Visualization
        curve = pyqtgraph.PlotDataItem(x=x, y=y)
        scatter = pyqtgraph.PlotDataItem(x=x, y=y + 2, pen=None, symbol="o")
        bars = pyqtgraph.BarGraphItem(x=x, height=y + 1, y0=-3, width = 0.15)
        self._plot.addItem(curve)
        self._plot.addItem(scatter)
        self._plot.addItem(bars)
        # Setup window content
        self.setCentralWidget(self._plot)
        self.setWindowTitle("PyQtGraph Demo")
        self.show()
```

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

4.1.2.1. 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 a 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 ore 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 show how mulitple 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.

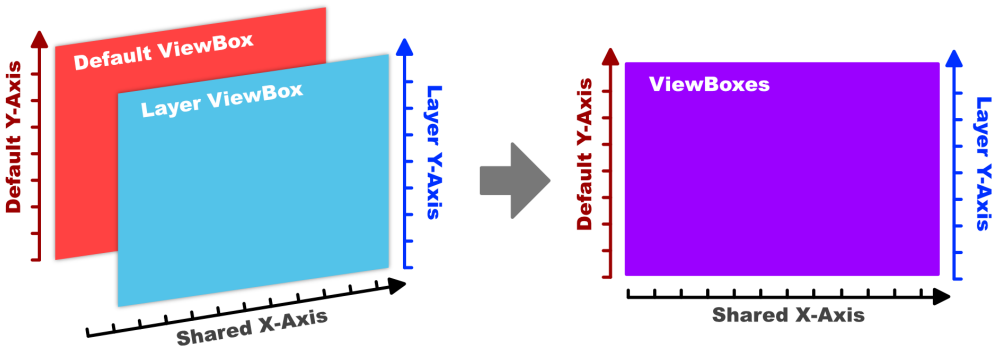

Abbildung 4.1.: Schema displaying different layers in a single plot.

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 continously 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 loggin 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 responsiblities 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. 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 4.2 shows the communication between these different layers. Operations marked in red are the intial setup of the connection between the components. Blue operatoins show the flow of data published from a process which is supposed to be visualized.

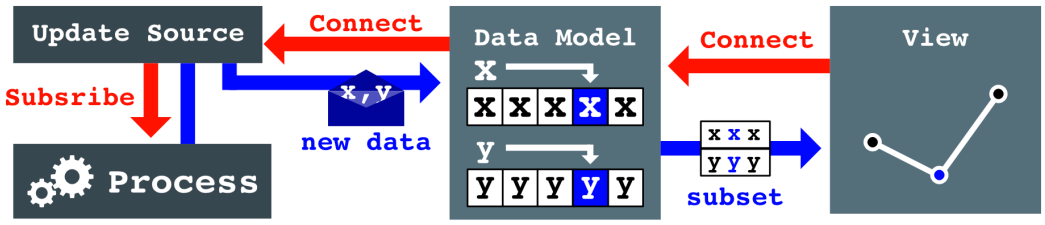

Abbildung 4.2.: Live Plotting Architecture for PyQtGraph. Operations for the initial connection are marked red, the data flow for new data is marked blue.

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. Design

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.

4.2.1. Architecture

...

4.2.2. Benchmarking Mechanism

...

4.2.3. Profiling

...

4.3. Implementation

4.3.1. Implementation of the Framework

...

4.3.2. Implementation of the Use Cases

...

5. Performance optimization

5.1. Optimizing line graph performance

5.1.1. GPU accellerated 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 Deghaye 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 Challange*. 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] Allessandro 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

```
import sys
from qtpy import QtWidgets


class MainWindow(QtWidgets.QMainWindow):

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

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

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

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

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


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

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

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


class ChangingLabel(QtWidgets.QLabel):

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


class LabelMouseClickFilter(QtCore.QObject):

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

```
            return True
        return super().eventFilter(o, e)


class MainWindow(QtWidgets.QMainWindow):

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

    def _install_filter(self, checked: int):
        if checked:
            self._app.installEventFilter(self._filter)
        else:
            self._app.removeEventFilter(self._filter)

    def _send(self):
        for label in [self._label_1, self._label_2]:
            event = QtGui.QMouseEvent(
                QtCore.QEvent.MouseButtonPress,
                QtCore.QPoint(0.0, 0.0),
                QtCore.QPoint(0.0, 0.0),
                QtCore.QPoint(0.0, 0.0),
                QtCore.Qt.LeftButton,
                QtCore.Qt.LeftButton,
                QtCore.Qt.NoModifier
            )
            QtCore.QCoreApplication.postEvent(label, event)

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    event_filter = LabelMouseClickFilter(parent=app)
    window = MainWindow(app)
```

```
73      sys.exit(app.exec())
```

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

A.2. PyQtGraph

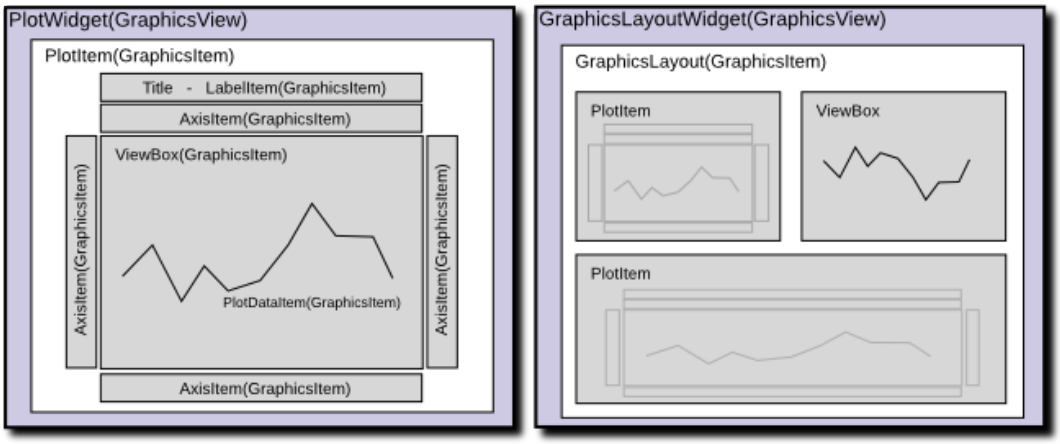

Abbildung A.1.: Different items involved in a PlotWidget.
Quelle: http://www.pyqtgraph.org/documentation/plotting.html

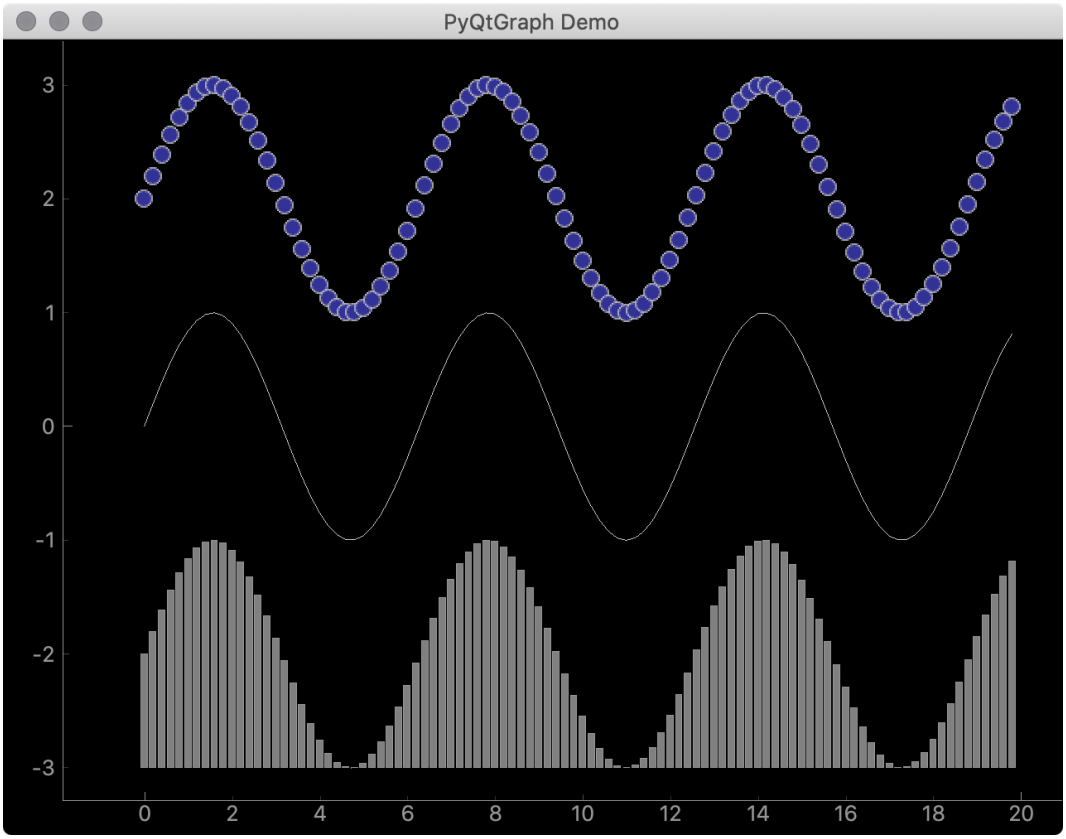

Abbildung A.2.: Window containing a plot created with PyQtGraph.

A.3. Matplotlib

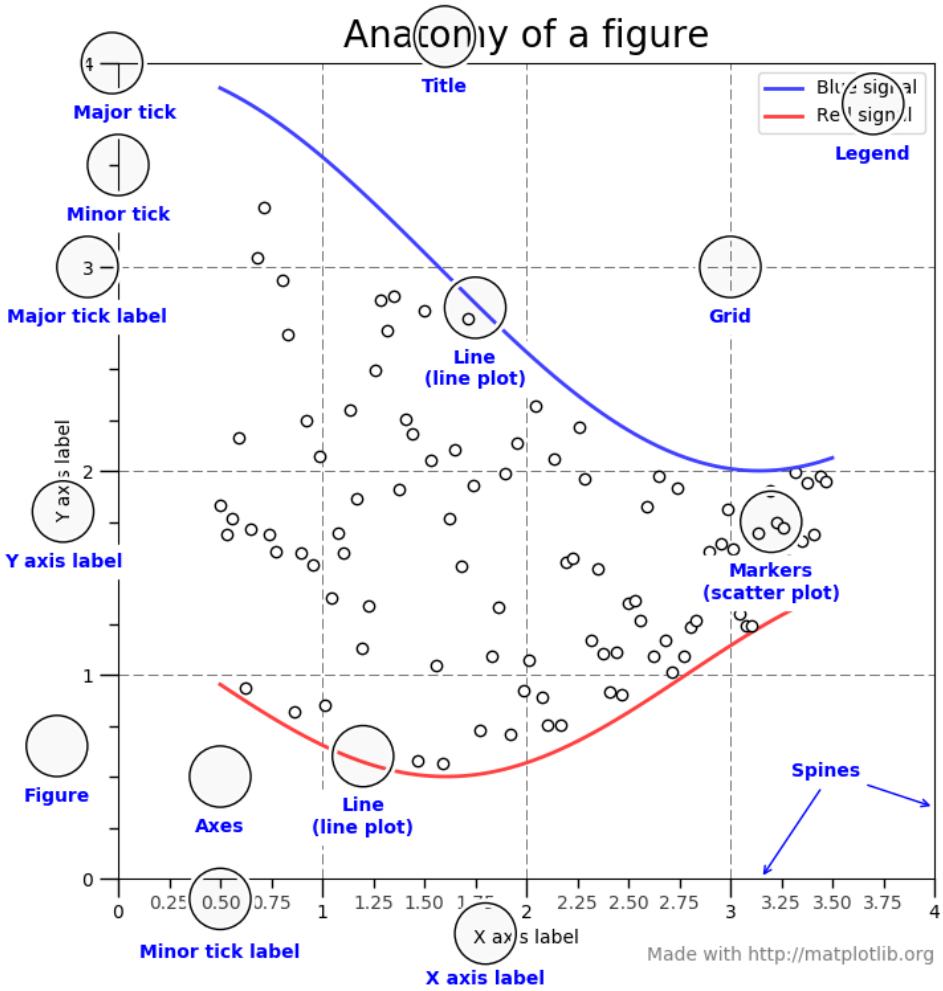

Abbildung A.3.: Different items involved in a Figure.
Quelle: https://matplotlib.org/3.1.1/tutorials/introductory/usage.html

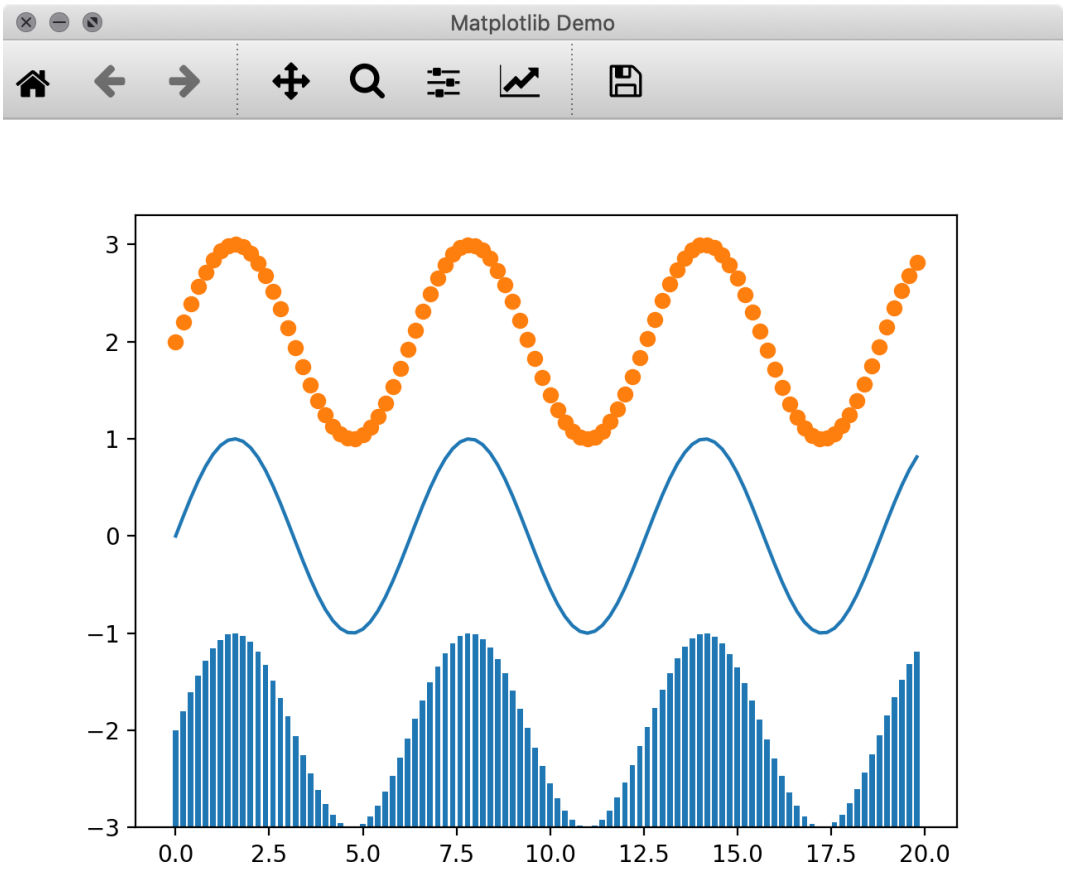

Abbildung A.4.: Window containing plot created with Matplotlib.