



LUND UNIVERSITY

Master Thesis

Development of a Graphical User Interface For Easier
Overview of Historical Chromatography Data

by

Agnes Hermansson



Department of Chemical Engineering
Lund University
Sweden
June 17, 2021

Supervisor: **Doctor Niklas Andersson**
Examiner: **Professor Bernt Nilsson**

Front page picture: Final GUI design

Postal address
PO-Box 124
SE-221 00 Lund, Sweden
Web address
www.lth.se/chemeng

Visiting address
Naturvetarvägen 14

Telephone
+46 46-222 82 85
+46 46-222 00 00

© 2021 by Agnes Hermansson. All rights reserved.

Printed in Sweden by Media-Tryck.

Lund 2021

Acknowledgement

Dedicated to my grandmother, Elly, who passed away on May 18th 2021.

I would like to thank my supervisor, Niklas, for supporting me during this project.

A thank you to Bernt and his research group for providing me with this master thesis. It has been a steep learning curve and hard work but also a lot of fun!

Thank You!

Abstract

Chromatography is a separation technique frequently used in the purification of medicinal drugs. In studies and project at the department of chemical engineering at Lund University a large amount of data have been generated through the use of their research software Orbit and their ÄKTA chromatography system. Methods have been developed for visualization of this data but there is currently no good way to have an overview of the data in their database. The aim of this master thesis was to develop a graphical user interface (GUI) in order to provide an overview of historical data generated in earlier projects as well as develop new methods for visualization and analysis. The final version of the GUI consists of four main parts: 1) A list widget with the possibility to sort and work with data, 2) A search bar for querying a particular set of data, 3) Tabs, nine in total, with windows for plotting, 4) Push buttons for plotting data and clearing the plot windows. A pop-up window with an overview of a run can be generated by double-clicking an item in the left-hand list of the GUI. A simpler design was also considered during this project but was discarded as it became more difficult to sort between runs. The final version of the GUI is, in large part, ready to be used but further improvements in terms of general stability needs to be considered for optimal functionality. Three new plot methods for visualizing total pool area and volume as well as a method for cycle analysis in periodic counter-current chromatography (PCC) were developed. The cycle plots gave results of varying quality; some runs showed a regular cycle patterns whereas none could be found in others. In particular, the plots for total pool area and volume could give a misleading impression due to a few large jumps in volume or area between runs, making it look like there is little or no variation between the lower values.

Sammanfattning

Kromatografi är separationsteknik som ofta används för upprening av läkemedel. I studier och project gjorda vid institutionen för kemiteknik vid Lunds universitet har en stor mängd data genererats genom användningen av deras forskningsmjukvara Orbit samt ÄKTA Chromatography Systems från Cytivia. Metoder för visualisering av denna data har tagits fram i tidigare projekt men i dagsläget finns inget enkelt sätt att få en överblick över datan i deras databas. Syftet med detta examensarbete var att utveckla ett grafiskt användargränssnitt (GUI) som kan ge en överblick över data som genererats i tidigare projekt samt att utveckla nya metoder för att visualisera och analysera data. Den sista versionen av GUI't består av fyra delar: 1) En uppsättning av två listor där man kan sortera mellan körningar (data från ett enda experiment) och välja att plotta denna datan, 2) En sökruta som söka efter en specifik körning, 3) Flikar, totalt nio stycken, som har fönster för plottar, 4) Tryckknappar för att plotta data och rensa plottfönstrena. Ett pop-up fönster kan genereras genom att dubbelklicka en artikel i den vänstra listan. En enklare version av GUI't togs också fram under projektets gång men den idén kasserades eftersom den designen gjorde det svårare att sortera mellan körningar och blev mer förvirrande att använda. Den slutgiltiga version av GUI't är till stora delar färdig att använda men förbättringar kommer behöva göras, bland annat i form av generell stabilitet för att fungera optimalt. Tre nya metoder för att visualisera den totala poolningsarean och poolningsvolymen togs fram samt en metod för att analysera återkommande cykler i *periodic counter-current chromatography* (PCC). Den senare metoden gav resultat av varierande kvalitet. Vissa körningar visade tydligt regelbundna cykler medan andra kunde uppvisa oregelbundna mönster. Plottarna för den totala poolningsvolymen och poolningsarean kunde ge ett vilseledande intryck eftersom magnituden på värdena mellan körningarna var ganska stora och de körningar med låga värden såg ut som de knappt ändrade sig.

A New Way of Interacting with Chromatography Data

With no easy way of getting a bird's eye view of the experimental data generated in previous chromatography projects at the department of chemical engineering at Lund University. This master thesis ventured out to put together a tool to fix this issue.

A graphical user interface (GUI) was developed with the aim of compiling the earlier work done in the area of chromatography at the institution into a single tool. By extracting chromatography data from the MongoDB database the user is able to look at historical data as well as data being uploaded in real-time (by updating the GUI). The final product is able to manage sets of data, plotting the same data in a variety of ways, as well as searching the database. Additionally, three new methods for visualizing the experimental data was created in order to provide the user with more options of analysis. The idea behind developing this GUI was to make it easier and quicker to visualize and analyse large amounts of data in the form of plots, tables and figures. This GUI also tackles the problem of not knowing how to code yourself, i.e. using a GUI requires absolutely no prior knowledge of any programming language in order to use it. Previously, the only way only way to look at the plots available in the GUI was to either run a script of code or create the methods yourself. At the moment of writing, the ability to analyze and process data through the GUI is literally just a few keystrokes away. Furthermore, the relevance of creating a GUI is also evident when it comes to research. Now, researchers can focus on the task of analysing data instead of trying to invent ways to visualize it since the GUI already provides the user with this. It is quicker to work with large amounts of data using the GUI, freeing up time for other areas of focus. Hopefully, this will come in handy in that it frees up time for other areas of research. Visualizing and analyzing data is a very important research task and should not be regarded lightly, but any tools that can automate routine parts of this work are always a convenient addition. The GUI was developed using the framework PyQt5, the NoSQL database MongoDB as well as the research software Orbit which has been developed for controlling chromatography systems at the department of chemical engineering at Lund University.

Ett nytt sätt att interagera med kromatografisk data

Innan detta projektet påbörjades fanns det inget smidigt sätt att titta på datan som uppstått i kromatografiska experiment vid institutionen för kemiteknik vid Lunds universitet. Det var detta som examensarbetet gav sig i kast med att lösa genom att skapa ett grafiskt användargränssnitt.

Ett grafisk användargränssnitt (kallas kort för GUI efter engelskans *graphical user interface*) utvecklades med syftet att sammanställa resultat och data från tidigare projekt som genomfört kromatografiska experiment och ur detta skapa ett enda verktyg för visualisering och analys. Genom att extrahera kromatografisk data ur *MongoDB* databasen så kan användaren titta på både historiska data så väl som data som laddas upp i realtid i databasen (om GUI't uppdateras kontinuerligt). Den sista versionen av GUI't klarar av att hantera och sortera mellan körningar (data som genererats av ett enda experiment), skapa grafer samt söka i databasen efter specifika körningar. Utöver detta så skapades tre nya visualiseringsmetoder i syftet att förse användaren med fler verktyg för analys av data. Tanken bakom detta GUI var att göra det enklare att få en överblick över den stora mängd data som ligger lagrad i databasen. Detta främst genom olika typer av grafer. Innan GUI't skapades så var man själv tvungen att gå in i ett skript med kod och köra det från någon typ av konsol eller att till och med skriva koden själv. Det är inte speciellt görbart om man t.ex. inte har tillräckliga förkunskaper i ett programmeringsspråk. Även i de fall då personen i fråga har tillräckliga förkunskaper så är tidsåtgången stor om man ska göra det själv. GUI't tar bort denna faktor eftersom den redan har metoder för att hämta den typ av analysverktyg man är intresserad av. Man behöver bara trycka på några knappar för att visualisera den data man är intresserad av. Detta gör det också betydligt snabbare att gå igenom stora mängder data under en kortare tid. Förhoppningsvis så kommer detta innebära att användaren kan lägga sin tid och energi på andra forskningsområden. Visualisering och analys av data är en central del i forskningen och ska inte förringas men det finns också ett behov att automatisera processen kring detta. Det är kanske där GUI't gör störst inverkan - att slippa uppfinna hjulet på nytt varje gång man vill titta på data. Användargränssnittet har blivit utvecklat i PyQt5, programmeringsspråket Python, forskningsmjukvaran Orbit samt med hjälp av *MongoDB* databasen.

Contents

1	Introduction	1
1.1	Aim	1
2	Background	2
2.1	Motivation	2
2.2	Chromatography	3
2.2.1	ÄKTA Systems	4
2.2.2	Orbit	5
2.3	MongoDB	5
2.4	GUI frameworks for Python	8
2.4.1	PyQt5	8
2.4.2	TkInter	8
2.4.3	Others	8
3	Material and Methods	9
3.1	Material	9
3.2	Method	9
3.2.1	PyQt5	9
3.2.2	Methods for retrieving data from MongoDB	12
3.2.3	Methods for plotting	12
3.2.4	Code Structure	13
3.2.5	New plot methods for data analysis	14
3.3	Work Flow	16
4	Result and Discussion	17
4.1	The GUI	17
4.1.1	Running the GUI	19
4.1.2	Development of the GUI	20
4.2	Data Visualization	22
4.2.1	Total pool area and volume	23
4.2.2	Cycle analysis of periodic counter-current chromatography	24
5	Future Work	26
6	Conclusion	28
	References	29

A	Appendix A	31
A.1	Main Window	31
B	Appendix B	34
B.1	Tabs	34
C	Appendix C	37
C.1	Pop-Up	37
D	Appendix D	38
D.1	New methods developed during the project	38

1 Introduction

In the process of making medicinal drugs a purification step is needed in order to remove unwanted compounds that can be hazardous or reduce the effect of the drug. In this master thesis the purification process of choice is liquid chromatography which is a separating technique based on the difference in equilibrium constants for different compounds. The department of chemical engineering at Lund University have developed a Python-based research software named Orbit for controlling the chromatography system provided by ÄKTA Systems. Orbit enables the user to create purification processes and add control logic directly in software. The software contains methods for sampling data, running sequences sequentially or in parallel, as well as communicating and synchronising with other orbit processes. Orbit also contains a database where data generated in the purification process is stored. MongoDB is the database provider and communication with the database is done through the Python package pymongo. At the outset of this master thesis there were no simple and quick way to get an overview of the data generated in the purification process but this idea was to develop a graphical user interface, also known as a GUI, in order to more readily being able to interpret and process large amounts of data. A GUI is an interface which allows the user to interact with it through graphical elements and these days there are several toolkits available for developing one. PyQt5 is a widely used and well documented GUI toolkit provided by *Riverbanks Computing* and was used to create the GUI for this master thesis.

1.1 Aim

The first aim of this master thesis was to study how data generated in the earlier studies and projects should be processed and presented in order to give the best possible overview of the generated data. The second aim was to investigate how a GUI could be designed and created in order to meet the needs of the first aim, which was the main focus for this master thesis.

2 Background

2.1 Motivation

The initiative for this master thesis comes from earlier projects where data have been generated and visualized in different manners. The main part of the data processing and visualizing comes from two separate chromatography set-ups: The first set-ups is a implementation of a high performance liquid chromatography system in a preparative chromatography platform which creates data called '*sobi2plus*' in MongoDB [15]. The set-up for this process can be seen in Figure 2.1.

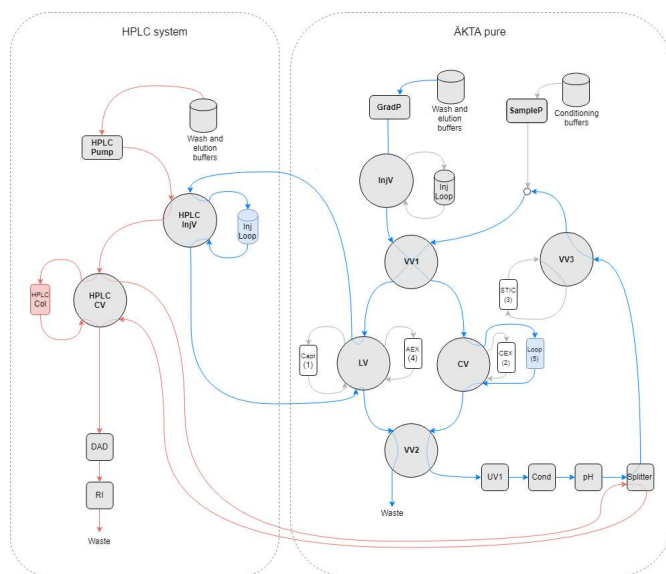


Figure 2.1 – Set-up for the implementation of a HPLC system into a preparative chromatography system. [15]

The second set-up is an optimization study on periodic counter-current chromatography (PCC) integrated in a monoclonal antibody downstream process [4]. Figure 2.2 shows the layout of the process.

This master thesis also builds on an earlier project aimed at visualizing the data in the database by creating new plot methods as well as complementary methods for

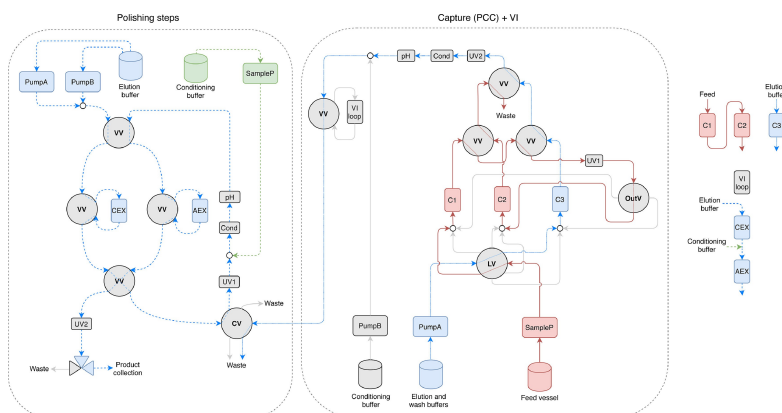


Figure 2.2 – The set-up for the integrated periodic counter-current chromatography process [4]

extracting historical process data [3]. Six new methods for visualizing data through plots were created and integrated into the class `Database` in the research software `Orbit`. In Figures 2.3a - 2.3d, these plots are briefly described.

For more information of the remaining two plotting methods, please refer to the project report [3].

2.2 Chromatography

Chromatography is a technique for separating, identifying and quantifying a component in a mixture [6]. A chromatography system consists of two phases, the mobile phase and the stationary phase. The mobile phase is a mixture of solvent in liquid chromatography whereas the stationary phase usually is a solid or a liquid coated on a solid support. This method of separation is based on a difference in equilibrium constants for the components in the mixture. The components will move at different speeds through the column due to their varying affinity for the stationary phase and therefore elute at different points in time. The mobile phase is set in contact with the stationary phase either by percolating the mobile phase through the stationary

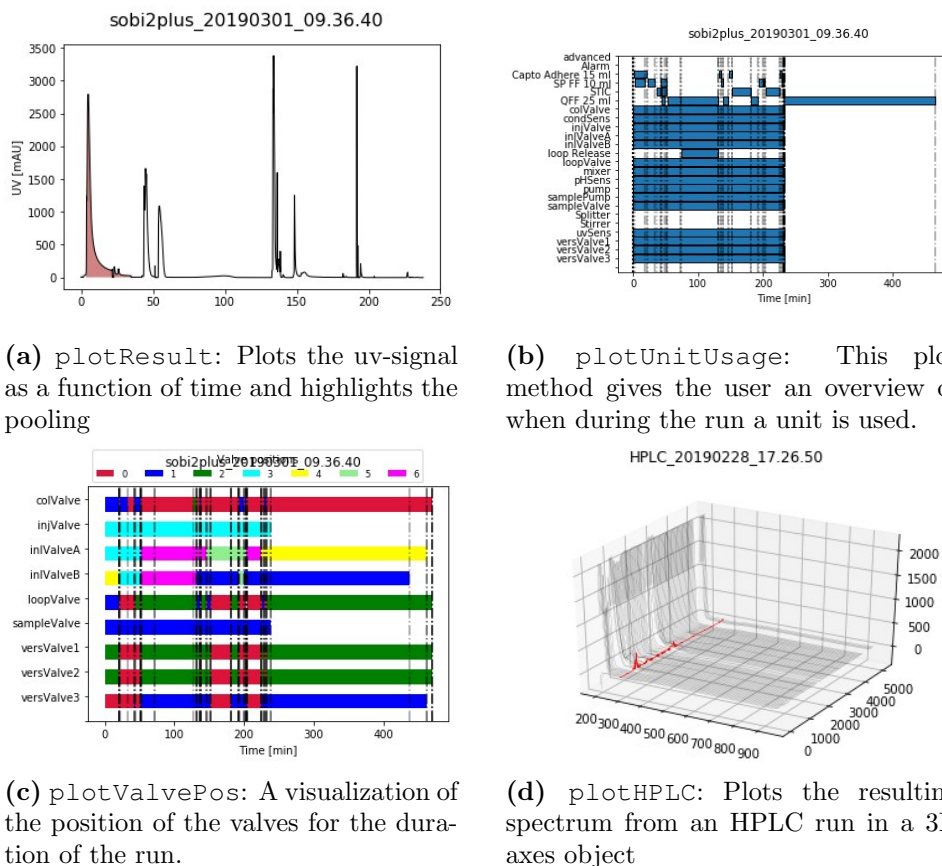


Figure 2.3 – The output from the methods developed in an earlier project with the aim of visualizing the data available in MongoDB.[3]

phase or by interaction in the column where the stationary phase is attached on the walls of the separation column or capillary.

2.2.1 ÄKTA Systems

The ÄKTA system is a chromatography system from Cytiva for purification of proteins, peptides and nucleic acids in small scale [2]. It comes with a built-in software called UNICORN™ which itself contains for main modules: Administration, Method Editor, System Control and Evaluation. UNICORN™ contain several pre-made methods for controlling the real-time chromatography system but also includes an extension system to allow users to add and change these or create entirely new ones.

2.2.2 Orbit

Orbit is a Python software developed at the department of Chemical Engineering at Lund University for the control of a chromatography system [9]. It was developed due to the need of a more flexible and powerful programming environment than what UNICORN™ could provide. Orbit could successfully be implemented since UNICORN™ offers a library that allows communication with the Unicorn server, enabling communication with the ÄKTA hardware with a custom software solution.

Orbit is made up of three parts: the interfaces, the unit library, and the system configurations. The main part of orbit is the `interfaces` file. It contains the `System` class which is the basis for communicating with the ÄKTA machine, as well as the classes `Units` and `Ports`. The `Units` class is used to create units for the chromatography system such as pumps and valves whereas the `Ports` class is used to connect units through tubes.

Experiments can be designed by creating new methods or by re-using existing ones. The use of dot notation enables the user to use methods from different parts of Orbit and the user can specify how the experiment should be executed. If specifying that it should connect to the database the data generated in the experiment is inserted into MongoDB.

2.3 MongoDB

MongoDB is a document-oriented NoSQL database with a range of different features [8]. In this project it served as a database for the data generated by the scripts run by Orbit as well as chromatography runs from the ÄKTA system. The database consists of three tiers where the databases is the first tier, collections the second and documents the third. These three tiers together make up a cluster. MongoDB has a range of different methods for retrieving, inserting and editing data in the database which has been incorporated in Orbit's database file. Communication with MongoDB is done via the Python module `pymongo` which offers a native interface to the operations listed above [16]. Orbit's database file implements this during the initiation of the Orbit class `Database`.

The class `MongoClient` is imported from the `pymongo` module. `MongoClient`

needs a host (named `address` in the code extract below) and a port as arguments to be able to connect to the database. The name of the database is also established as well as the name of the collections.

```
1 class Database(object):
2     def __init__(self, address='localhost', port=27017):
3         '''
4         Connects to the database
5
6         address    - ip address to the mongodb database server
7         port       - port number of the mongodb database server
8
9         '''
10        self.dbname = 'orbit'
11        self.client = MongoClient(address, port)
12
13        self.db = self.client[self.dbname]
14
15        # Add orbit collections
16        self.collections = ['Runs', 'Signals', 'Buffers', 'Units',
17                            'Ports', 'Data', 'FlowPaths', 'RunLogs', 'Tubes']
```

When a connection has been established it is possible to work with the data present in the database. In the database file of Orbit there are methods for adding and removing data which are based on pymongo methods. In order to add data to the database, the methods in the database file make use of the two pymongo methods `insert()` or `insert_one()`. In the code extract below an example of `insert()` is given, the only difference between the two methods is that the former can add more than one document to a collection. The two methods both need an document to insert as argument.

```
1 def insertData(self, orbitTime, dataDict):
2     '''
3     Inserts a data item in the collection 'Data'
4     '''
5
6     item = dataDict
7     item['Time'] = orbitTime
8     item['RunId'] = self.runId
9
```

```
10 self.insert('Data',item)
```

Orbit's database file also contains methods for removing documents from the MongoDB which are based on the method `remove()`. It takes the document to be removed as argument, see example below.

```
1 def removeResult(self, runId):
2     """
3     This method removes the run with the given runId from the
4     database.
5     """
6     collections = ['Signals', 'Buffers', 'Units', 'Ports', 'Data', '
7     FlowPaths', 'RunLogs', 'Tubes']
8     for collection in collections:
9         self.db[collection].remove({'RunId':runId})
10    self.db['Runs'].remove({'_id': runId})
```

It is also possible to query the database using a filter as argument with the `find_one()` or `find()`.

```
1 def loadResult(self, run):
2     """
3     Loads a result given a run
4     """
5
6
7     runId = run['_id']
8
9     result = {}
10
11
12    for collection in self.collections:
13        tic = time.time()
14        items = [x for x in self.db[collection].find({'RunId':runId
15        })]
16        result[collection] = items
17        etime = time.time()-tic
18
19        result['etime'] = etime
```

```
20     result [ 'Runs' ] = run
21
22     return result
```

The methods mentioned here are only a small selection of methods available, for more examples see [8].

2.4 GUI frameworks for Python

To develop a graphical user interface in Python there is a plethora of choices and below a small selection of common frameworks are presented.

2.4.1 PyQt5

PyQt5 is a collection of Python bindings to the fifth version of the Qt application framework and consists of a set of a set of Python modules [14]. Qt consists of a set of C++ libraries and development tools for creating both graphical user interfaces, GUIs, and application programming interfaces, APIs. Qt also offers a software tool called Qt Designer which allows drag and drop functionalities, making it significantly easier to create a GUI without much prior coding knowledge. However, this tool was not investigated in this project.

2.4.2 TkInter

TkInter is considered to be the standard python GUI package and is a thin object-oriented layer on top of Tcl/Tk [5]. The main advantages of TkInter is that it is fast and that it is included in the Python standard library.

2.4.3 Others

PySide2 is a more recent binding to the Qt toolkit and is developed by the *Qt Company* [17]. wxPython is a cross-platform GUI toolkit and through Python module extensions, make use of the wxWidgets C++ library. [11]

3 Material and Methods

3.1 Material

The material used in this master thesis has been developed in the Python programming language using the development environment provided by Spyder 3.7, the GUI toolkit PyQt5 (version 5.9.2), MongoDB 4.2 and the Orbit database systems. In this project the main form of interaction with MongoDB was through the graphical user interface *MongoDB Compass*.

The first part of this project was done on a Windows 10 laptop with an Intel Core i3-5010U @ 2.10 GHz with 4GB RAM and the second part of the project was done on a Windows 10 Pro desktop computer with an Intel Core i7-6700K @ 4.99 GHz with 20 GB RAM.

3.2 Method

3.2.1 PyQt5

PyQt5 is made up of eleven modules, each containing classes with different functionalities. The main module used in this project was the QWidgets module which makes up the core of the development of the GUI. Each of the classes used in the GUI are presented in Table 3.1 [13].

Table 3.1 – List of used widget classes and its methods, each of which is reached using the method call syntax (i.e., the dot operator).[13]

Widget class	Method	Description
QHBoxLayout		Creates a layout where each element added to it are ordered horizontally.
	addWidget	Adds a widget to the layout in a horizontal order.

	addLayout	Nests another layout under QHBoxLayout.
QVBoxLayout	addWidget	Creates a layout where each element added to it are ordered vertically.
	addWidget	Adds a widget to the layout in a horizontal order.
	addLayout	Nests another layout under QHBoxLayout.
QGridLayout	addWidget	Creates a layout where each element added to it are ordered in a 2D grid. Enables the programmer to control where each widget should be placed.
	addWidget	Adds a widget to the layout in a horizontal order.
	addLayout	Nests another layout under QHBoxLayout.
QListWidget	addItem	Creates an item-based list.
	addItem	Adds an item to the list.
	count	Returns the number of items in the list.
	currentItem	Returns the marked item in the list.
	currentRow	Holds the current row.
	insertItem	Inserts an item at the given row.
	item	Returns the item that is in the current row in the list.
	selectedItems	Returns a list of all selected items in the list.

	<p>setCurrentRow Set the given row as the current.</p> <p>takeItem Returns and removes the item from the list.</p> <p>clear (slot) Removes the list of all items.</p> <p>itemDoubleClicked A signal is emitted when an item in the list is double-clicked.</p> <p>itemSelectionChanged A signal is emitted when a new item in the list is selected.</p>
QLineEdit	<p>textChanged A signal is emitted whenever the text to the widget changes.</p>
QPushButton	<p>clicked.connect Connects the pressing of the button with an action .</p> <p>setStyleSheet Depending on given input this function sets the appearance of the push button.</p>
QCheckBox	<p>isChecked Querys wether the checkbox is checked, returns a boolean.</p>
QTabWidget	<p>addTab Adds a tab to the widget.</p>
QTableWidget	<p>setRowCount Sets the number of row of the table.</p> <p>setColumCount Sets the number of colums for the table.</p>

	setItem	Sets an item for a given row and column.
QTableWidgetItem		Creates items for QTableWidgetItem.

3.2.2 Methods for retrieving data from MongoDB

MongoDB has a range of commands to use for retrieving, inserting and editing data present in the database. The orbit database script are based of these methods for handling data and was incorporated in to the development of the GUI. Mainly three methods in the database file that were used for retrieving data and they are presented in Table 3.2.

Table 3.2 – List of the used methods for retrieving data from the Orbit database files.

Method	Output
getRuns	Querys the database given the inputs and returns all matching runs in a list.
loadResult	Returns a dictionary for a given run with the collections as keys and list of data from the collection as the values.
getPoolTimes	Returns a list of tuples where each tuple contains the start and stop times for the pooling.

3.2.3 Methods for plotting

A large part of the graphs presented in the GUI were developed in a previous project [3]. They were available as a part of the Orbit database file and are described in Table 3.3. Additionally, three new plots were generated from the data present in MongoDB during this project which are presented in the ”*Results and Discussion*” section.

Table 3.3 – List of implemented methods for plotting data from MongoDB, each of which were created in an earlier project [3].

Method	Output
plotResult	Plots the uv-signal as a function of time.
plotValvePos	Generates a horizontal bar graph with the valve type on y-axis with colors indicating a certain valve position.
plotUnitUsage	Generates a horizontal bar graph with the unit type on y-axis with blue-colored bars indicating unit usage.
plotHPLC	Generates a 3D plot from the resulting spectrum of a HPLC run
plotTogether	A custom plot method that creates a graph to showcase three parameters in one figure uv-signal, column usage and phase changes.
plotColumnwise	Generates a plot that visualizes the result from individual columns on separate axes. Can be plotted with or without time gaps.

3.2.4 Code Structure

The structure of the code for the GUI has been divided up into four scripts: 1) Main Window, 2) Tabs, 3) PopUp, and 4) Database. Each script have its own specific function and the primary motivation for this design was to improve the code readability and the response time when interacting with the GUI. In the text that follows, each of the four scripts that make up the GUI are explained.

1. **Main Window** - The main window script is where the visual design of the GUI is created as well as connecting actions, such as pressing a push button, with the right event.
2. **Tabs** - The Tabs script consists of nine classes where one class makes up one tab. Each tab contains a method for extracting plots from Orbits database file as well as a method for clearing the plot windows in the GUI.
3. **PopUp** - The Pop-up script is a class that is being called when an item in the left list of the GUI is double-clicked. A pop-up window is generated which presents the user with information about the run in the form of plots and a table with information available in the Runs collection.
4. **Database** - The database file came together with the orbit folder that was received in the beginning of this project. It contains methods for retrieving data from the MongoDB as well as most methods for plotting the same.

The outline of the code can be found in Appendix A - C.

3.2.5 New plot methods for data analysis

New plot methods were created to provide the user with more tools for visualizing data from MongoDB. In the following two sections, methods for visualizing the total pool area and volume are described as well as a method for detecting reoccurring cycles in periodic counter-current chromatography. The full code for the methods created here can be found in Appendix D.

Periodic counter-current chromatography

The data from the optimization study on periodic counter-current chromatography (PCC) integrated in a monoclonal antibody downstream process has two *uv*-signals in the database, *uv1* and *uv2nd*. A method was developed to provide the user with a view of how cycles in the PCC data repeats themselves in a single run. The cycles are set to start when the elution phase begins and ends before the next elution phase starts. Two methods were created in order to do this: The `get_cycledata` and

`plot_cycles`. The first method takes the result of a run as an input argument and extracts the times when the elution occurs and returns it as a list. `plot_cycles` also takes the result of a run as an argument and then creates a plot of the cycles from this information together with the list of elution times from `get_cycledata`. The uv-signal plotted is the *uv2nd*-signal. The reason this uv-signal was chosen and not the *uv1*-signal was because the latter signal did not have the same repeating pattern as the former one. The resulting plots can be seen in Figure 4.7. Another method was also generated for the same sort of data. However, it was an almost copy of the method developed in an earlier project, `plotResult`, with the difference being which type of uv-signal was set as the default.

Total pool volume and area

First, the two methods for calculating the total pool volume, `calcPoolVolume`, and area, `calcPoolArea`, were created. They both take the output from the Database methods `loadResult` and `getPoolTimes` as arguments and return the sum of the pool volume and pool area respectively. The total pool area was estimated using the `scipy` method `trapz(y, x)`, which integrates along the given y-axis using the composite trapezoidal rule [10]. The integration range was computed using the pool times and a Boolean index array. The total pool volume calculations was based on the simple equation given below:

$$V_{pool} = q \cdot t_{pool} \tag{3.1}$$

where q is the volumetric flow rate and t_{pool} is the pool time. The flow rate was received in the same way the integration axis was in `calcPoolArea`. Two new methods for plotting the total pool volume (`plotPoolVolume`) and pool area (`plotPoolArea`), each of which both takes a list of results of a run and the corresponding list of pool times as arguments, were created. Figure 4.6 gives a plot example. Inside each function there is a line of code that uses the calculation methods described above to compute the total pool volume and pool area respectively. The final result can be seen in Figure 4.8.

3.3 Work Flow

The GUI has been designed in stages and several versions have been created during the project. Starting out, the first part of this project consisted of studies on PyQt5 and MongoDB. In section 4.1.2, Figure 4.4a, the first iteration of the GUI can be seen. Prior to this, studies on how to retrieve and insert data in MongoDB had been conducted as well as building very simple user interfaces [12]. This period also contained some initial experimentation with Orbit. Orbit came along with some premade examples of experimental runs, which was looked studied in order to understand how data was generated and how Orbit interacted with MongoDB. After the first period of initial studies on the subject of GUI frameworks, Orbit and MongoDB, the focus was now on reviewing data generated in earlier projects as well as implementing methods for plotting this data into the GUI. A project done two years earlier had created six new plot methods and these were all integrated and the plots can be drawn in the plot windows of the GUI [3]. Due to the large amount of data that was received from earlier projects, a new widget for managing runs was implemented. This is also were two different versions of the GUI were created. In section 4, Figures 4.1 and 4.5 these two version are presented. It was also during this period that a new layout was designed for the GUI. Since the GUI steadily grew it was necessary to divide up the GUI into subsections of some sort. This was done by introducing tabs with `QTabWidget`, see Table 3.1.

Lastly, three new methods for plotting data were developed: Two methods for plotting the total pool area and volume as well as one for plotting cycles occurring in the data from the optimization study on periodic counter-current chromatography [4]. It was also now that it was decided on what the final design would be, see Figure 4.1.

4 Result and Discussion

4.1 The GUI

The final version of the GUI is presented in Figure 4.1. Earlier iterations can be seen in Figure 4.4 and Figure 4.5. The GUI is made up of two parts, a lower layout and an upper layout. The upper layout contains two list widgets for working with runs which are a data set from an experiment, a search bar and the push buttons for moving runs between lists as well as push buttons for clearing and plotting run data. The lower layout contains nine tabs where each of which is connected to a different type of plot.

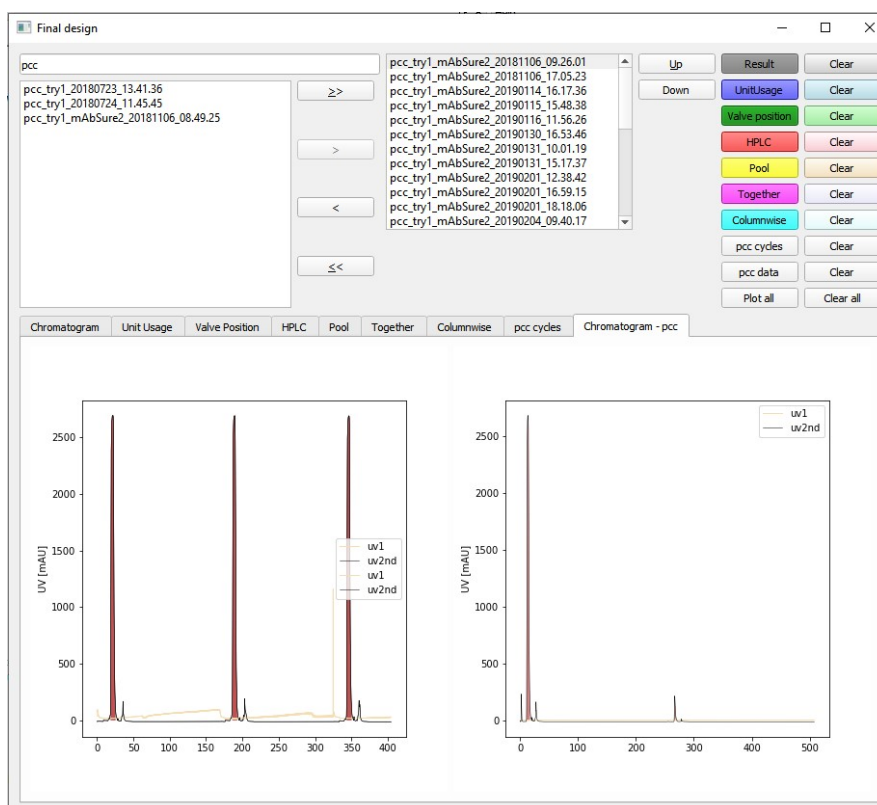


Figure 4.1 – Final design.

The Search Bar

The search bar in the upper part of the GUI consists of the widget `QLineEdit` coupled with a search method called `searchfunction`. The search method takes a string as input argument and sees if there is a match in the collection `Runs`, and if that is the case, the items will be added to the left-hand list. If the search bar is given no input, no runs will be added and if given the keyword *all*, every run currently in MongoDB will be added.

The Lists

`QListWidget` creates the two lists in the upper part of the GUI. The design and functionalities of the two lists comes from a tutorial [7]. The items in the left list can either be moved to the right list by using the arrow buttons situated between the two lists. Additionally, it is possible to double-click the item and launch a pop-up window giving information about that single run. Figure 4.2 shows an example of such a pop-up window. The items in the right list are available for plotting and it is the first two items in this list that will be shown when pressing the plot buttons.

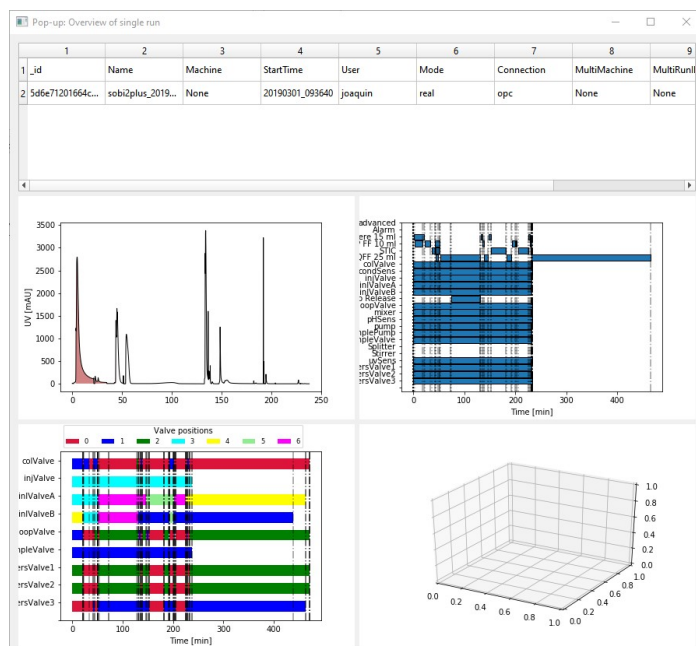


Figure 4.2 – A pop-up window for a run in the database. It is generated when double-clicking an item in the left-hand list of the GUI.

Plot Buttons and Tabs

The push buttons in the upper right corner are the widgets called `QPushButton`. They are connected to a method for extracting plots in the `Tabs` file using the signalling interface exposed by `clicked.connect()`.

There are nine tabs in total, each of which is a separate class with its own layout and methods for retrieving and plotting data from the database as well as methods for clearing the plot windows in the GUI.

4.1.1 Running the GUI

The pressing of one of the plot buttons triggers a series of events in the GUI code. This section aims to explain how the GUI works beneath the surface. Figure 4.3 gives a rough sketch of what happens when a push button is being pressed. The GUI is started by running the `Main Window` file (see section 3.2.4) in the Spyder console.

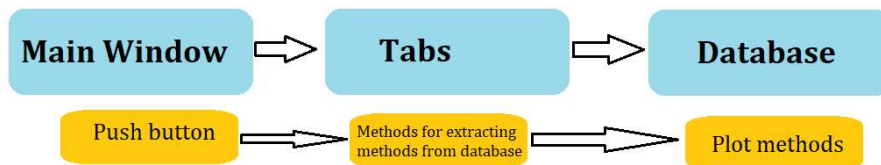


Figure 4.3 – A visual overview of what happens when a plot button on the interfaces is pressed. The blue boxes represent the code files whereas the orange boxes represent the line of action when a push button is being pressed.

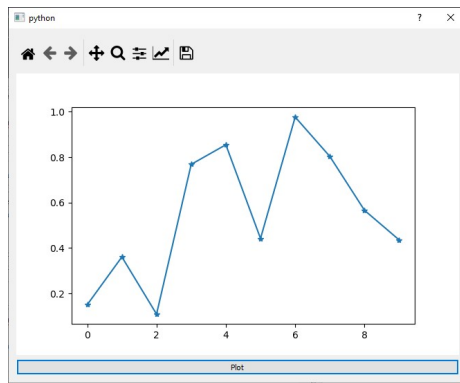
There are, in large part, three tiers in this GUI. The `Main Window` contains the surface level actions which then delegates the task of plotting to the two other tiers. The second tier is the `Tabs` file, here methods for calling and drawing the plots of the plot methods from the `Database` class have been developed. The third tier is the `Database` class and it contains all methods for plotting, calculations, such as pool area and volume, and methods for extracting data from MongoDB. The plot buttons in the GUI are all connected to a method in the `Main Window` file that calls the methods in the `Tabs` file. The signal `clicked.connect()` connects the action of pressing a button with the event of a plot being drawn.

The action of double-clicking an item in the left list widget and generating the pop-up window is more or less the same as pressing a push button with the exception that the signal `itemDoubleClicked.connect()` was used instead.

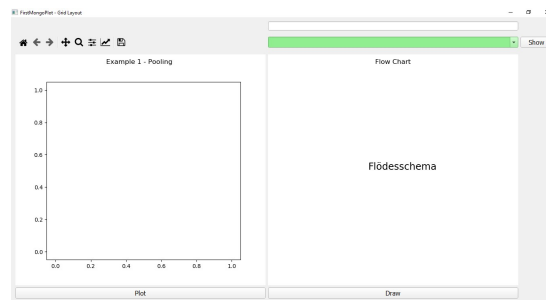
4.1.2 Development of the GUI

The very first version of this GUI was a simple design with a plot window and a button to show the result of a run and is shown in Figure 4.4a. This purpose of this first iteration was to educate myself on the basics GUI development using PyQt5 and getting a first glimpse of what the final product might look like. With this as a starting point, I could start thinking about what the GUI should contain and what data it should be able to present. Knowing that the GUI should be able to provide the user with the functionality of comparing runs it was decided that the GUI would need at least two windows for plotting as well as some kind of overview of the runs present in the database. The GUI then evolved into containing two set of plot windows and a widget called `QComboBox` which had a toggle list with the names of the runs, each of which was coupled with a push button. Figure 4.4b shows this version of the GUI. The next version of the GUI took a big leap and was developed into a design with a list widget and tabs. In order to give the user a better overview of the runs available in the database a search function was also added. At this stage it was only the name of the run that was searchable. At this point, another GUI was designed in parallel to try out various interface options to figure out which would be the best in the end. Alternative 1 is the design used for the final version of the GUI, shown in Figure 4.1, and alternative 2 is presented in Figure 4.5. The only difference between the GUIs is the upper layout. Alternative 2 has a simpler layout: A list widget with an overview of the runs available in the database were each run is clickable and several if the items in the list can be selected for plotting. Alternative 1 is a set of list widgets with push buttons between them, enabling items to be moved from one list to the other. The left list gives the current runs and the right list enables the user to plot them. The two different alternatives share the same search function which was developed from the first version. It is now possible to search the keywords present in the `Runs` collection in MongoDB.

In large part, alternative 1 and 2 are very similar. Two different options were developed in order to use both and see which would be the easiest to work with. Initially, alternative 2 were thought to be the best option since it entails less code



(a) Iteration 1: A simple user interface with a push button for plotting



(b) Iteration 2: Two plot windows with a QComboBox, a search bar (at this point in time without a connection to a method), and two push buttons for plotting.

Figure 4.4 – The two first versions of the GUI

than alternative 1. This makes it easier to launch the GUI and also to debug when necessary. However, users quickly discovered that having only a single list with a multiple choice option makes it more difficult to keep track of which runs one is currently working or is interested in. This problem is solved in alternative 1 since the runs of interest can be transferred to the right list.

The pop-up functionality was created to give the user to have a better overview of a particular run. It is worth noting that the content of the current pop-up window is in a very a rough initial stage and may need some further development to be truly useful. The idea behind it was to give the users a quick overview of an individual run whereas the main GUI would be focused on a comparative view. The final GUI has two graphs per tab which is the minimum amount required to be able to compare runs. Also, the GUI was initially developed on a small screen computer making it inconvenient to have more than two graphs at once. It is however easily redesigned

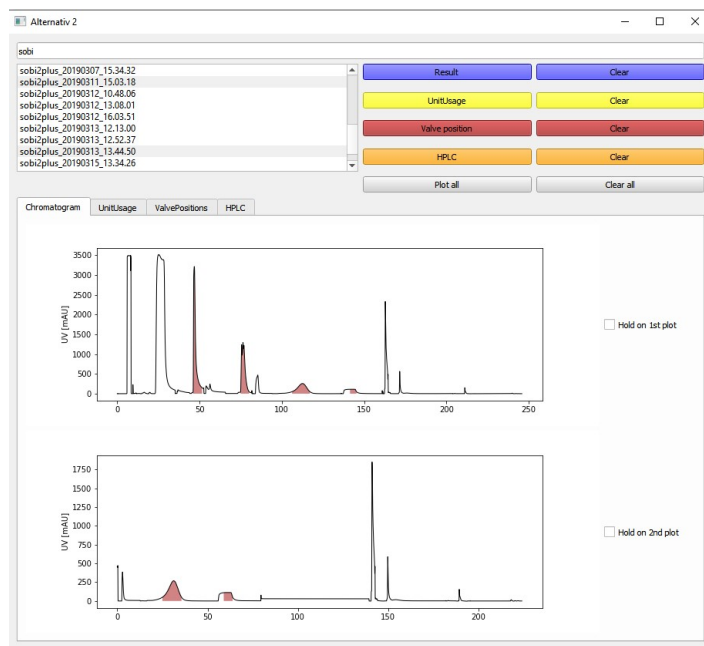


Figure 4.5 – This design version of the GUI was shortly investigated during this master thesis but was discarded as it confusing to keep track of which runs were worked with.

to add more plot windows to the GUI. The decision to include tabs into the GUI was taken after reviewing other designs from an unpublished presentation [1]. In this GUI the designer had chosen to have different pages which were switched between using toggle list instead of tabs. This was considered to be more of a hinder to the workflow than using tabs and was therefore not investigated. The use of push buttons to plot and clear the graphs was at the outset the only idea considered but at the later stages it became evident that it would make the look of the GUI more confusing, especially since the GUI might continue to grow in the number of push buttons needed. A possible solution to this would be to have one set of push buttons; one for plotting and one for clearing, and connect them to the widget `QComboBox` were all the plot functions could be incorporated.

4.2 Data Visualization

This project also aimed at processing and visualizing data present in the database. Three new plot methods were created together with three complementary methods for extracting and calculating the necessary input to the new plot methods. First, methods for calculating and plotting the total pool area and total pool volume

were developed and thereafter methods for analysing cycles occurring in periodic counter-current chromatography were created.

4.2.1 Total pool area and volume

The main problem with these plots is when there is a big difference in values between runs. In Figure 4.6 there is a large jump for three of the runs for the volume plot and one for the area plot making it hard to interpret the actual difference between runs, especially considering that the runs with relatively low values of total pool volume and total pool area appears to change very little. An attempt was made at breaking up the y-axis in order to make it easier compare the runs with relatively small values of the total pool volume or total pool area, but was not able to reach a simple solution within the timeframe of this master thesis. It is also questionable whether this way of visualizing the total pool area and volume is the best. In hindsight, a better option would probably be to present the values in a table instead of a plot. Especially, since the values differ a lot in terms of magnitude. It could also be argued that it is the method that is faulty but this was not looked at due to time limitations.

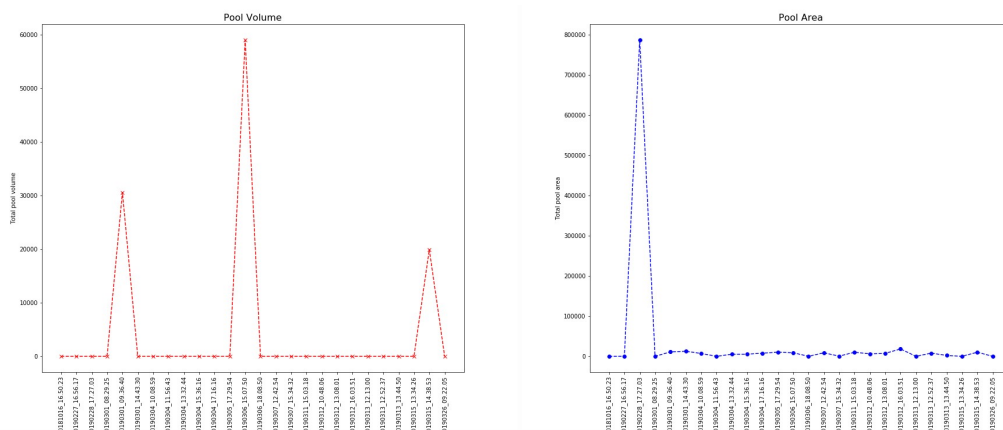


Figure 4.6 – The methods for plotting the total pool area (blue) and volume (red). The total volume or area is plotted against the name of the run.

4.2.2 Cycle analysis of periodic counter-current chromatography

The output from the method for visualizing the cycles occurring in the data from the PCC runs gave promising results. The method of starting the cycles at the time of elution works well in most cases but for some of the runs in the database there is no simple pattern to be found. This can be down to the simple form of the method or that the experiment has been carried out in a non-optimal way or that the runs is only a test run and therefore not complete.

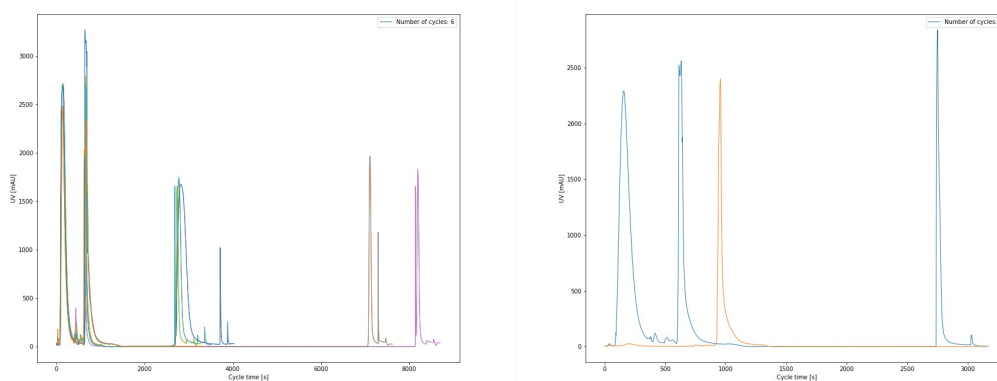


Figure 4.7 – Graph of two runs using the method developed for analysing cycles in period counter-current chromatography.

Below, in Figure 4.8, the final result of the method copied from an earlier project is shown [3]. The reason it was created was to get a better understanding of the PCC data and was actually done before creating the method for visualizing the cycles. When trying to plot the PCC data using the `plotResult`, which has the `uv1` - signal as the default signal, the highlighted pool area would not follow the signal at all, and changing the default signal from `uv1` to `uv2nd` made the highlight fit the peaks.

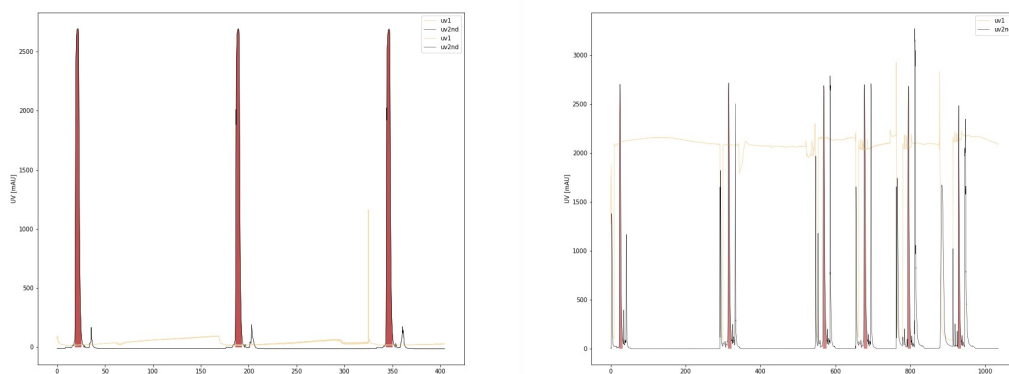


Figure 4.8 – Graph of two runs using the method developed in an previous project in order to visualize where the pooling starts and stops [3].

5 Future Work

Early on in the project, the possibility of working with live data via MongoDB was shortly investigated. The `watch()` method in MongoDB lets the user know whenever a change happens to a collection of interest. However, it was quickly discovered that creating a GUI for real-time data analysis would require significantly more time than what was feasible within the scope of this master thesis. Thus, the project mainly focused on historical data. However, a rough solution to this problem is simply to update the GUI during a run. Since the GUI extracts data from the database in real-time it would give a close approximation of a live plot.

There are improvements to the design of the GUI could be made: For example, the amount of push buttons steadily increased as more plots were incorporated into the GUI and might give a messy impression. This could be solved by rearranging them into a `QComboBox`, which is a combination of a button and a pop-up list, together with only two `QPushButtons`, one for plotting and one for clearing. Also, at the time of writing, there is no simple way of knowing when an error has occurred when working with the GUI. The user have to manually look at the console window for potential error messages. In the future it would be practical to be able to receive these error messages in either a pop-up window or maybe a console on the GUI itself. Toolbars could also be added to the plot windows of the GUI in order to provide the user with a tool to zoom in and out of figures and being able to save figures in the GUI.

The pop-up window is still in an early development phase and could be further improved. One suggestion is to introduce the double-click functionality to the right-hand list. In addition to this, there is more work to be done regarding the general stability of the GUI.

There is one problem that surfaced early on in the project that was not solved during this project. When launching the GUI a lot of `matplotlib` figures are spawning. I don't know why that is, only that it should be fixed by the command:

```
1 import matplotlib
2 matplotlib.use('Qt5Agg')
```

This worked at first, but when Orbit's Database class was integrated with the GUI, multiple figures started spawning when launching the GUI.

6 Conclusion

The main reason this master thesis was decided to be done in PyQt5 and Python is two fold: The first being that Orbit was only available in this programming language and choosing another language would not make sense. Secondly, PyQt5 was suggested as the preferred GUI toolkit when first starting out. It also quickly became evident along the course of the master thesis that it was the easier GUI toolkit to work with due to the abundance of tutorials on the Internet.

The final version of the GUI consists of four main parts: A search function which queries the collection *Runs* in MongoDB for a match, a set of lists which gives the user the tool to sort between and plot runs, push buttons for plotting and clearing the plot windows, as well as nine tabs each of which have two plot windows and a plot method associated with them. In addition, it is possible to double-click a list item in the left-hand list of the GUI to generate a pop-up window. The pop-up window gives the user a quick overview of a single run using a table with the information from the collection *Runs* as well as the four plots from an earlier project [3]. The pop-up as it is today is still in need of some further development to be truly useful however. The GUI developed in this project gives the user a good tool for visualizing and working with data in MongoDB. There is a functionality for searching the database as well as a list widget for sorting out and working with data of interest. Additionally, multiple choices of plotting methods are available for data visualization.

Three new plot methods were developed for visualizing how the total pool area and volume changed between runs as well as a method for analysing cycles in periodic counter-current chromatography (PCC). The former plots allow a user to quickly identify runs where the pool area and volumes varies significantly. The plots of the PCC cycles gave a mixed result in terms of regularity as some of them showed an obvious repeating pattern whereas others proved harder to interpret.

References

- [1] Anita Solbrand, Mats Nilsson, Christer Eriksson, Magnus Bergman, Mikael Berg, Pikkei Wistrand-Yuen. *Janus*. Accessed: 2021-03-12.
- [2] Cytiva. *ÄKTA pure Chromatography Systems*. <https://cdn.cytivalifesciences.com/dmm3bwsv3/AssetStream.aspx?mediaformatid=10061&destinationid=10016&assetid=16276>. Accessed: 2021-06-06.
- [3] Daniel Espinoza. “Automatic visualization of data from protein purification processes”.
- [4] Joaquín Gomis-Fons, Niklas Andersson, and Bernt Nilsson. “Optimization study on periodic counter-current chromatography integrated in a monoclonal antibody downstream process”. In: *Journal of Chromatography A* 1621 (2020), p. 461055. ISSN: 0021-9673. DOI: <https://doi.org/10.1016/j.chroma.2020.461055>. URL: <https://www.sciencedirect.com/science/article/pii/S0021967320302673>.
- [5] *Graphical User Interfaces with Tk*. <https://docs.python.org/3/library/tk.html>. Accessed: 2021-06-06.
- [6] Georges Guiochon and Oliver Trapp. “Basic Principles of Chromatography”. In: *Ullmann’s Encyclopedia of Industrial Chemistry*. American Cancer Society, 2012. ISBN: 9783527306732. DOI: https://doi.org/10.1002/14356007.b05_155.pub2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/14356007.b05_155.pub2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/14356007.b05_155.pub2.
- [7] Jie Jenn. *How to move items between QListWidgets — PyQt5 Tutorial*. March 2021. URL: https://www.youtube.com/watch?v=tH7_QFuyX48.
- [8] *MongoDB*. <https://www.mongodb.com/>. Accessed: 2021-06-06.
- [9] Niklas Andersson, Anton Löfgren, Marianne Olofsson, Anton Sellberg and Bernt Nilsson. *The Orbit Controller*. Department of Chemical Engineering, Lund University. 2016.
- [10] *numpy.trapz*. 2021-06-04. URL: <https://numpy.org/doc/stable/reference/generated/numpy.trapz.html>.
- [11] *Overview of wxPython*. <https://www.wxpython.org/pages/overview/>. Accessed: 2021-06-06.
- [12] *PyQt5 tutorial :Learn how you can create a Python GUI in 2021*. <https://build-system.fman.io/pyqt5-tutorial>. Accessed: 2021-06-15.

- [13] *Qt Documentation*. <https://doc.qt.io/>. Accessed: 2021-06-01.
- [14] Riverbanks Computing. *What is PyQt?* <https://riverbankcomputing.com/software/pyqt/intro>. Accessed: 2021-06-06.
- [15] Simon Tallvod. “SOBI2+: Internal report”. 2020.
- [16] *Tutorial*. <https://pymongo.readthedocs.io/en/stable/tutorial.html>. Accessed: 2021-06-06.
- [17] *Tutorial*. <https://pymongo.readthedocs.io/en/stable/tutorial.html>. Accessed: 2021-06-06.

A Appendix A

A.1 Main Window

```
1
2
3
4 class MainWindow(qtw.QWidget):
5     def __init__(self):
6         super().__init__()
7         '''H r skapas GUIs layout '''
8
9     def UpperLayout(self):
10
11    def LowerLayout(self):
12
13
14
15 ##### BUTTONS FOR LIST WIDGET #####
16
17    def updateButtonStatus(self):
18
19    def Add_1_toList(self):
20
21    def Remove_1_fromList(self):
22
23    def Add_All_toList(self):
24
25    def Remove_All_fromList(self):
26
27    def buttonUp(self):
28
29    def buttonDown(self):
30
31
32
33 ##### BUTTONS FOR PLOTTING #####
34
```

```
35
36     def button_plotResult(self):
37
38     def button_plotUnitUsage(self):
39
40     def button_plotHPLC(self):
41
42     def button_plotPool(self):
43
44     def button_plotTogether(self):
45
46     def button_plotColumnwise(self):
47
48     def button_plotCycles(self):
49
50     def button_plot_pccdata(self):
51
52     def button_PlotAll(self):
53
54
55 ##### BUTTONS FOR CLEARING #####
56
57     def button_clearResult(self):
58
59     def button_clearUnitUsage(self):
60
61     def button_clearValvePos(self):
62
63     def button_clearHPLC(self):
64
65     def button_clearPool(self):
66
67     def button_clearTogether(self):
68
69     def button_clearColumnwise(self):
70
71     def button_clearCycles(self):
72
73     def button_clearPccData(self):
74
75     def button_clearAll(self):
76
```

```
77
78
79     def searchfunction(self, text):
80
81     def connectButtons(self):
82
83         # Connect buttons
84
85     def launchPopUp(self):
86
87
88 # Kalla applikationen
89 app = QtWidgets.QApplication([])
90 mw = MainWindow()
91 app.setStyle(QtWidgets.QStyleFactory.create('Fusion'))
92 app.exec_()
```

B Appendix B

B.1 Tabs

```
1 class FirstTab(qtw.QWidget):
2     def __init__(self):
3         super().__init__()
4
5         # Layout f r fliken
6
7     def get_plotResult(self, runs):
8
9     def clearResult(self):
10
11
12 class SecondTab(qtw.QWidget):
13     def __init__(self):
14         super().__init__()
15
16         # Layout f r fliken
17
18     def get_plotUnitUsage(self, runs):
19
20     def clearUnitUsage(self):
21
22
23
24 class ThirdTab(qtw.QWidget):
25     def __init__(self):
26         super().__init__()
27
28         # Layout f r fliken
29
30     def get_plotValvePos(self, runs):
31
32     def clearValvePos(self):
33
34
```

```
35 class FourthTab(qtw.QWidget):
36     def __init__(self):
37         super().__init__()
38
39         # Layout f r fliken
40
41
42     def get_plotHPLC(self, runs):
43
44     def clearHPLC(self):
45
46
47 class FifthTab(qtw.QWidget):
48     def __init__(self):
49         super().__init__()
50
51         # Layout f r fliken
52
53     def get_plotPool_Volume_Area(self, runs):
54
55     def clear_Poolplots(self):
56
57
58 class SixthTab(qtw.QWidget):
59     def __init__(self):
60         super().__init__()
61
62         # Layout f r fliken
63
64     def get_plotTogether(self, runs):
65
66     def clearTogether(self):
67
68
69 class SeventhTab(qtw.QWidget):
70     def __init__(self):
71         super().__init__()
72
73
74         # Layout f r fliken
75
76     def get_plotColumnwise(self, runs):
```

```
77
78     def clearColumnwise(self):
79
80
81 class EighthTab(qtw.QWidget):
82     def __init__(self):
83         super().__init__()
84
85
86         # Layout f r liken
87
88     def get_plot_cycles(self, runs):
89
90     def clearCycles(self):
91
92
93 class NinthTab(qtw.QWidget):
94     def __init__(self):
95         super().__init__()
96
97
98         # Layout f r liken
99
100    def get_plot_pccdata(self, runs):
101
102    def clearPccData(self):
```

C Appendix C

C.1 Pop-Up

```
1 class PopUp(qtw.QWidget):
2     def __init__(self):
3         super().__init__()
4         # Creating the layout of the Pop-up
5
6
7     def fill_table(self, run):
8
9     def Pop_plot(self, run):
```


D Appendix D

D.1 New methods developed during the project

```
1 def calcPoolArea(self, result, PoolTimes):
2     '''
3     En funktion som ber knar poolarean och detta ska g ras mha av
4     trapz-funktion.
5     '''
6     data0 = [[item['Time']/60., item['uv1']] for item in result['
7 Data']] if 'uv1' in item]
8
9     if not data0:
10        return 0
11
12    data = np.array(data0)
13    t = data[:,0]
14    uv = data[:,1]
15
16    pool_area = list()
17    for start, stop in PoolTimes:
18        idx = (t >= start) ^ (t > stop)
19        local_t = t[idx]
20        local_uv = uv[idx]
21        area = trapz(local_uv, local_t)
22        pool_area.append(area)
23    return sum(pool_area)
```

```
1
2
3 def calcPoolVolume(self, result, pooltimes):
4     '''
5     PoolVolume = Fl det * Pooltiden
6     Enhet?
7     '''
8     # if not 'FlowPaths' in result:
9
10    data0 = [ [item['Time'], item['Flow']] for item in result['
11 FlowPaths']]
```

```

12     if not data0:
13         return 0
14     data = np.array(data0)
15     t = data[:,0]
16     flow = data[:,1]
17
18     pool_volume = list()
19     for start, stop in pooltimes:
20         idx = (t >= start) ^ (t > stop)
21         local_t = t[idx]
22         local_flow = flow[idx]
23         for i in range(len(local_t)):
24             pool_volume.append(60*local_t[i] * local_flow[i])
25     return sum(pool_volume)

```

```

1
2
3     def plotPoolVolume(self, results, pooltimes, ax1 = None):
4
5         volume = [self.calcPoolVolume(results[i], pooltimes[i]) for i
6 in range(len(results))]
7         y_volume= np.array(volume)
8         x_names = [results[i]['Runs']['Name'] for i in range(len(
9 results))]
10
11         x = [i for i in range(0, len(volume))]
12
13         if ax1 == None:
14             fig= plt.figure()
15             ax1 = fig.subplots()
16
17         ticks = [i for i in range(0, len(volume))]
18         #print(ticks)
19         labels = [x_names[i] for i in range(len(ticks))]
20         plt.xticks(ticks, labels, rotation = 'vertical')
21         plt.ylabel('Total pool volume')
22         plt.title('Pool Volume', fontsize = 16)
23         ax1.plot(x, y_volume, 'rx—')

```

```

1
2
3     def plotPoolArea(self, results, pooltimes, ax2 = None):

```

```
4
5     area = [self.calcPoolArea(results[i], pooltimes[i]) for i in
range(len(results))]
6     y_area = np.array(area)
7     x_names = [results[i]['Runs']['Name'] for i in range(len(
results))]
8     x = [i for i in range(0, len(area))]
9
10
11     if ax2 == None:
12         fig = plt.figure()
13         ax2 = fig.subplots()
14
15     ticks = [i for i in range(0, len(area))]
16     labels = [x_names[i] for i in range(len(ticks))]
17
18     plt.xticks(ticks, labels, rotation = 'vertical')
19     plt.ylabel('Total pool area')
20     plt.title('Pool Area', fontsize = 16)
21     ax2.plot(x, y_area, 'bo—')
```

```
1     def get_cycledata(self, result):
2
3         elution_times = list()
4
5         for entry in result['RunLogs']:
6             if 'Info' in entry and 'phaseName' in entry['Info'] and
entry['Info']['phaseName'] == 'Elution':
7                 elution_times.append(entry['Time']/60)
8
9         return elution_times
```

```
1
2     def plot_pccdata(self, result, ax=None, plotpools=True):
3
4         data01 = [[item['Time']/60., item['uv1']] for item in result['
Data'] if 'uv1' in item]
5         data1 = np.array(data01)
6         #     print(data1)
7
8         data02 = [[item['Time']/60., item['uv2nd']] for item in result['
Data'] if 'uv2nd' in item]
9         data2 = np.array(data02)
```

```

10 #     print('data2 = ',len(data2))
11     if len(data1)>0 or len(data2) > 0:
12         if ax == None:
13             fig = plt.figure()
14             ax = fig.subplots()
15 #             fig.suptitle(result['Runs']['Name'], fontsize=16)
16
17             ax.plot(data1[:,0],data1[:,1], color='wheat', linewidth=1,
label = 'uv1')
18             ax.plot(data2[:,0], data2[:,1], color = 'k', linewidth=0.5,
label = 'uv2nd')
19             ax.legend()
20
21         if plotpools == True:
22             PoolTimes = self.getPoolTimes(result)
23             for entry in PoolTimes:
24                 i1 = next(i for i, val in enumerate(data2[:,0]) if
val > entry[0])
25                 i2 = next(i for i, val in enumerate(data2[:,0]) if
val > entry[1])
26                 ax.fill_between(data2[i1:i2,0],data2[i1:i2,1],
facecolor=(0.7, 0.2, 0.2), alpha=0.6)
27                 ax.set_ylabel('UV [mAU]')
28
29             else:
30                 print('No %s data in %s'%( 'uv2nd', result['Runs']['Name
']))

1
2
3     def plot_cycles(self, result, ax = None):
4
5         data02 = [[item['Time']/60.,item['uv2nd']] for item in result['
Data'] if 'uv2nd' in item]
6         data2 = np.array(data02)
7 #         print(data2)
8
9         if not data02:
10             return print(result['Runs']['Name'],'contains no uv2nd
signal')
11
12
13

```

```
14     elution_times = self.get_cycledata(result)
15 #     print(elution_times)
16
17     if not elution_times:
18         return print(result['Runs']['Name'], ' has no elution time')
19
20     cycles = [[]]*len(elution_times)
21     x = [[]]*len(elution_times)
22
23     for i in range(len(cycles) - 1):
24         cycles[i] = np.array([d[1] for d in data2 if d[0] >=
25 elution_times[i] and d[0] < elution_times[i+1] ])
26         x[i] = np.linspace(0, len(cycles[i]), len(cycles[i] ))
27 # Different for last iteration.
28         cycles[-1] = np.array([d[1] for d in data2 if d[0] >=
29 elution_times[-1]])
30         x[-1] = np.linspace(0, len(cycles[-1]), len(cycles[-1] ))
31
32     if ax == None:
33         fig = plt.figure()
34         ax = fig.subplots()
35         fig.suptitle(result['Runs']['Name'], fontsize = 16)
36 #         f = ['crimson', 'blue', 'green', 'cyan', 'yellow', '
37 lightgreen', 'magenta', 'lavender', 'wheat', 'chartreuse', 'brown', '
38 darkblue']
39 #         l = ['1st cycle', '2nd cycle', '3rd cycle', '4th
40 cycle', '5th cycle', '6th cycle', '7th']
41
42     n = len(elution_times)
43
44     for i in range(len(cycles)):
45         ax.plot(x[i], cycles[i], linewidth = 1)
46         ax.legend(['Number of cycles: {}'.format(n)])
47         ax.set_ylabel('UV [mAU]')
48         ax.set_xlabel('Cycle time [s]')
```