Report and Technical Documentation 23.09.2024

Plotting Refactoring Project for Sasview Julius Karliczek

Apr. 2024 – Sept. 2024

My work on the project started in April 2024 and addresses new features in SasView that can all be summarized under the term ‘plotting refactoring’.

There are some sources in the SasView GitHub that are related to this project. I will include them here at the beginning. There is also a label under which all the issues and pull requests for this project are listed. This label is ‘big refactoring project’. Another helpful label might ‘plotting’

Discussions:

Some general requirements for the refactoring project: <https://github.com/orgs/SasView/discussions/2475>

Regarding plotting packages: <https://github.com/orgs/SasView/discussions/2244>

Thoughts on plotting functionality: <https://github.com/orgs/SasView/discussions/2549>

Not closely related to this project, but was discussed in a meeting: <https://github.com/orgs/SasView/discussions/2951>

Pull Requests:

For my demo: <https://github.com/SasView/sasview/pull/2940>

For the integration of the demo: <https://github.com/SasView/sasview/pull/3111>

Main issue with the right label for this: <https://github.com/SasView/sasview/issues/2937>

Demo:

The standalone demo was supposed to give a first overview on what should be achieved in the six months of the project. Some features that the demo has right now are listed here, further explanations on which classes and methods are used for what are explained afterwards. Just testing out the demo can also simply be done by starting MainWindow.py in the subfolder ‘plotting\_refactor’ in the branch ‘plotting\_refactor\_tabs’ in the main SasView GitHub repository.

1. Plotting of a randomly created dataset with a corresponding fit in either 1D or 2D
2. Provide a central PlotWidget that bundles all the created plots for the internally existing datasets. (measurement/data/residuals)
3. A DataViewer that provides an overview of all the datasets andplots currently existing in the program
4. Adding modifiers to 1D (colors and linestyles) and 2D (color schemes) to plots by adding them via a button in the DataViewer and dragging them onto the plottable/plot with automatic update upon adding the modifier
5. Adding datasets from one plot to another one by dragging them across the DataViewer. This means that multiple lines from different plots in the program can all be shown in the same one.
6. Having multiple 1D or 2D plots in the same figure (the same SubTab with the same canvas) is interactive. Clicking on one of the figures on the side will process this and exchange the position so that it will become the big one

Most of the python files in the demo project have docstrings for their classes and their methods. But since that may not be enough, I will describe them here more in-depth.

I will begin with the MainWindow.py file, since it involves the code that starts the application and the MainWindow object also instantiates some of the other objects in its constructor that are central to the architecture.

MainWindow.py

The start of the application is invoked by the main method in MainWindow.py and first of all connects the excepthook method. This ensures that the errors from the Qt App will be shown in the IDE integrated terminal. Then, the QApplication for the demo is created as this provides useful interactions for the QWidget based demo, for example event handling or integrating user desktop settings from the operating system onto the now opening program. After that, the MainWindow is created, shown afterwards and then the QApplication is executed until the window is closed.

To get a design for the MainWindow in the application, we use the Qt Designer. This is an easy way to design a Qt Widget and generate a .ui file. This .ui file can be converted into a python file by a command line method that looks like this: “pyuic6 –o output.py –x input.ui”.

Once we have generated the python file for our designed widget, we can import the design of this widget by letting our internal class MainWindow be a child of the Ui\_MainWindow class that is defined in MainWindowUI.py. Ui\_MainWindow has the central method setupUi which can be used to get the design for the window from the designer directly applied to our window. This is done by calling “self.setupUi()” in the constructor of our MainWindow class.

Before that, there is also the call of the constructor of the normal QWidget QMainWindow, so that our object of MainWindow can be shown and support all the Qt features.

After that we set the window title and size in the constructor and declare the variable fitPageCounter to keep track of how many fitpages are existing in the fittingTabs QTabWidget. (fitPageCounter could surely be replaced from the perspective now, since fittingTabs.count() can directly give you the amount of tabs that are open in the tab widget right now.) If you have a closer look at the MainWindowUI.py file, you can see that we have nested a QTabWidget (fittingTabs) in a QWidget (centralwidget) in the MainWindow. FittingTabs will provide the needed structure for being able to plot multiple datasets at the same time without replacing it.

Some other things we have initialized in the MainWindowUI.py file are the menubar that can add a new fitpage via a button, two buttons on the bottom “calculate” and “plot” that can calculate the dataset with the provided numbers and settings from the FitPage and a button to show and hide the data viewer.

Then, the first FitPage is added to the TabWidget, the DataViewer is created and the control buttons for the functionality are connected to their respective methods.

For the calculation and the plotting of the datasets that are about to be created, when the buttons in the MainWindow are pressed, the identifier from the FitPage class is used to distinct, if this “signal” for calculation comes from an already existing FitPage that has to be recalculated or from a new one, where no simulated data is already saved in the program. In calculate, one can see that the checkboxes from the figpage are asked about their state and the information is then fed to the update\_dataset method of the dataviewer. Therefore, one can already see, that the dataviewer has to do something with the storage and handling of the internally created data.

Fitpage.py

The FitPage is also designed in the Qt designer and follows the same strategy to use this by subclassing the class from the generated python file and then setting up the design by setupUi in its constructor.

The elements in this class are: the identifier, which keeps track of an ‘id’ that this fitpage is identified with, three spinboxes that allow manipulation of the variables that are used to calculate some scattering data from the underlying formula, a combobox to decide which form factor is used in the calculation and two checkboxes that decide if the calculation provides a 1d or 2d scattering pattern and if a fit dataset should also be calculated.

The getter methods for this class are self-explanatory. Only the index\_changed method is used to see if the third combobox should be activated, because only the cylinder can be calculated with a height.

DataViewer.py

The DataViewer connects the functionalities from the demo.

Instances of other classes that the DataViewer owns are:

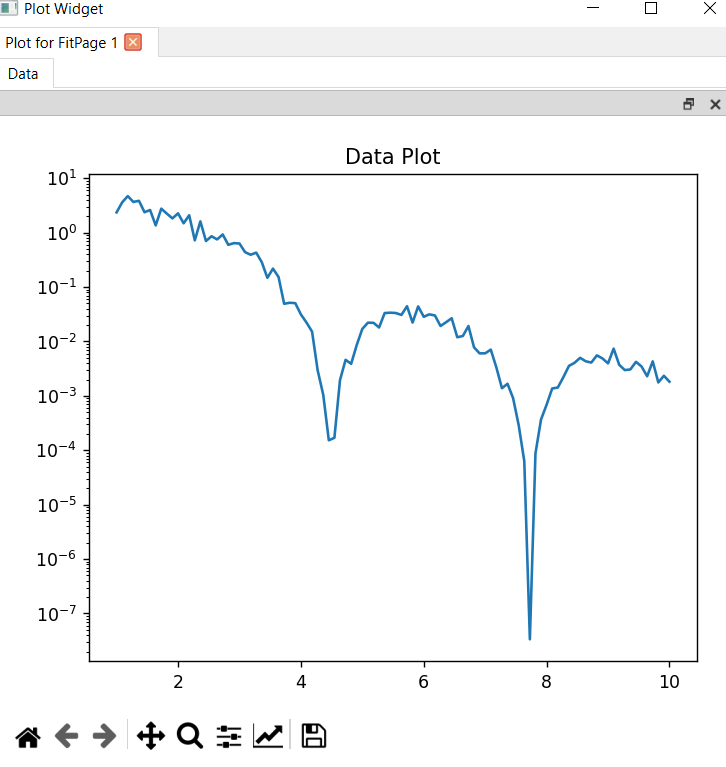
* DataCollector: keeps track and stores the already calculated data
* DataTreeWidget: shows the already created data visually in the DataViewer window
* PlotTreeWidget: shows the already existing plots visually in the DataViewer window
* PlotWidget: centralizes the plotting function for all plots
* ModifierCombobox: can add a modifier to the PlotTreeWidget that can be dragged onto a plottable or plot to modify them

The DataViewer also uses setupUi to use the design from the .ui file in its constructor.

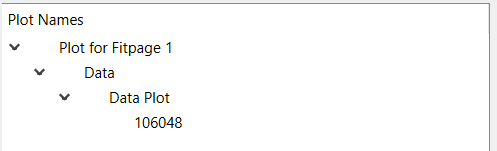
The plotting function is not direct, because the DataViewer does not communicate directly with the PlotWidget to tell it that it should start plotting some dataset from the DataCollector. Instead, the plotting works indirectly through the PlotTreeWidget and the communication is mostly one sided, since the information is fed to the PlotWidget, but existing plots are not tracked but only all of them are redrawn upon changes.

The first function to invoke this process is create\_plot that is called in onPlot in the MainWindow. It only calls the update\_plot\_tree method with the fitpage index it got earlier, shows the plotWidget and activates it afterwards. Hence, the further process is taken to update\_plot\_tree.

Update\_plot\_tree is a longer method and maybe needs some discussion, since the main structure for what is shown in the PlotTreeWidget and the PlotWidget comes from there. At first, all top level items from the PlotTreeWidget are considered and checked if they are a TabItem. TabItem in this case means, that this item should later represent a major tab in the PlotWidget under which multiple subtabs can be combined later. This checking is needed, because once modifiers are introduced to the PlotWidget, these do not need to be considered when the aim is to plot data. When iterating over all the top level TabItems, it is checked whether the underlying FitPage index for this TabItem is the same of the one that this method was invoked with. If this is true, it means that a plot for this FitPage index already exists and it needs updating. The updating is realized by deleting the TabItem with the same FitPage index from the tree and creating a new one afterwards that will represent the new plot.

If there is no TabItem with the same FitPage index, we can conclude that this FitPage has not been plotted earlier and a new sequence of items for this FitPage can be inserted. Instead of only inserting one item that is then representing everything that has been created in the plot widget for this FitPage, the sequence of items has one item for each tab, subtab, plot and line in a plot (in the case of 1d) which are subsequent children of the TabItem in the PlotTreeWidget.

To visualize this, the following screenshots can help:



One can see, that there are 3 subsequent children of the TabItem with the name “Plot for Fitpage 1”. When this is compared to the structure that is now shown from the plot in the PlotWidget, one sees some parallels. Every item in the PlotTreeWidget corresponds to another feature from the PlotWidget.

1. Plot for Fitpage 1 corresponds to the major tab in the PlotWidget
2. Data represents the SubTab
3. Data Plot represents the plot (this is the equivalent to a matplotlib figure)
4. 106048 represents the line on the figure

Since this analogy really helps with the architecture, because for the creation of all the major tabs, subtabs, matplotlib figures and lines inside of them, later, we only need to iterate over all the existing items in the PlotTreeWidget and create them correspondingly to what they are.

Notice that in case there is a 2d plot, a plottable (the lowest item on this ladder) will be marked with the indicator “2d”.

In order to get this architecture into the code, all the items in the PlotTreeWidget are subclasses of the QTreeWidgetItem and have properties to store values that they need to store. These classes can be seen in the PlotTreeItems.py file.

To get a better overview of what these subclasses are needed for, we will jump to PlotTreeItems.py.

PlotTreeItems.py

There are four classes “TabItem”, “SubTabItem”, “PlotItem” and “PlottableItem”. As mentioned before, these are representative for the four stages of the plotting process and where a plot for a certain fitpage needs to go so that it is displayed correctly.

Therefore, although it may not be good practice, because data and its representation in QWidgets can be more encapsuled by the model/view paradigm, the classes here have some properties.

TabItem contains everything that will be generated for one fitpage. Because of that, it also has the property fitpage index.

SubTabItem inherits TabItem and therefore needs a fitpage index and a subtab index for instantiation. The subtab index gives information about the nested tab widget index that the child items of this item need to be displayed in.

Next in the hierarchy is the PlotItem which inherits SubTabItem and needs a fitpage index, a subtab index, an ax index to know which axes the plottable children are displayed in and is\_plot\_2d to identify if this plot is filled with 2d data or not. This also determines if multiple plottables can be dragged onto this plot, because one plot can only display one 2d dataset.

The plottable item is representing a line in a plot and has the attributes data\_id and type\_num, which are used for identifying, which dataset in the datacollector belongs to this item and what kind of item it is, a data, a fit or a residual, respectively. The type\_num goes from 1 to six and identifies the dataset in the collector with either 1d or 2d and data, fit or residuals.

Going back to the DataViewer, now we can see how the items are added to the PlotTreeWidget in the right oder. First, a new TabItem is added for the current given fitpage\_index. This can be done in every case, because we deleted the TabItem from the tree if it already existed for the fitpage index.

The TabItem is created and takes the plotTreeWidget as its parent, because it is a top level item, a display name (which always needs to be in square brackets, because PlotTreeWidgets can have multiple columns) and the fitpage index. Now we need to deal with the problem that the return type of an item from the PlotTreeWidget will always be just a QTreeWidgetItem and not the specified subclasses that were discussed earlier. To work around this, the subclass itself can be saved in the QTreeWidgetItem in the tree with set data and then the fitpage index can be retrieved again by for example “tabitem.child(index).data(0,1)”. The return type of this statement will be a SubTabItem and then the property methods can be used again.

Next, the first subtabitem and plotitem are created, since this is needed in every case. Then the data\_id of the dataitem for this fitpage index is collected, so that it can be linked when the plottables are created in the next step.

For creating the plottables, we check if there is a 2d or a 1d dataset, name it accordingly and give it the right type\_num. This will be important in the real plotting process later.

If the dataset does not contain a respective fit, we are done here. If not, fit and residuals need to be displayed, too. This is done in 2 separate extra tabs, one contains only data and fit and the other one includes the residuals, too. Notice, that in the 2d case we need one plot for every plottable.

After populating the PlotTreeWidget, the plotting process can start. This is done in redraw. The name and the function parameters already indicate, that all the plots for in a certain tab will be redrawn with respect to the current items in the PlotTreeWidget. The redraw method is also connected to the drop Signal of the plotTreeWidget, so that when a Plottable from one Plot is dragged to another Plot to display it there, the tab is also redrawn.

In redraw, the parameters are the redraw\_fitpage\_index and the redraw\_subtab\_index, to redraw all tabs and specify, which tab and subtab should be activated, once the redrawing is complete.

To redraw every tab, we iterate through the top level TabItems in the PlotTreeWidget and call the redrawTab from the PlotWidget with the current TabItem. The next steps of the process are discussed under the PlotWidget section.

To keep the PlotTreeWidget consistent in its items, a function for clearing the according top level item on closing the tab in the plot widget is needed. That is done by the remove\_plottree\_item function. Since the order in which the fitpages are plotted is not always the same, (e.g. the Fitpage 1 can be plotted third, Fitpage 2 first and Fitpage 3 second.) we need to ask the datacollector, which fitpage the data that is displayed in this tab belongs to. When we found out about the fitpage index, we can iterate through all the top level items and take delete the right one.

The next two methods are for the modifiers that can be added to the PlotTreeWidget. setupModifierCombobox adds the items to the combobox, which directly have the right naming for matplotlib in the part after the equal sign.

The modifiers are also classes that can store their own object in the QTreeWidgetItem so that it can be retrieved later when needed in the plotting.

The other functionality of the DataTreeWidget are rather simple. This is handled in update\_dataset and update\_datasets\_from\_collector. The update\_dataset method is invoked once the calculation or the plot button is pressed in the MainWindow. It directs the call for creating or updating a dataset to the datacollector. When the dataset is in the right state, the upper DataTreeWidget in the DataViewer UI can be updated, which is done in update\_datasets\_from\_collector. To only add datasets that are new in the datacollector, we iterate through all items and check if an instance for the fitpage\_index already exists in the DataTreeWidget. Then we add a PlotPageItem for the Fitpage that this dataset comes from, one DataItem for the data itself and if a fit exists in the DataCollector, a representing item is also added.

The scheme of having QTreeWidgetItems representing the data from the DataCollector is similar to above with the PlotTreeWidget. These classes are stored in DataTreeItems.py.

DataTreeItems.py

The PlotPageItem extends the QTreeWidgetItem and saves itself in it with the saveData method. Additionally, the fitpage index and the data\_id for the dataset from the DataCollector are saved.

The DataItem extends the PlotPageItem and adds the type\_num attribute, which again represents if this dataset in the dataviewer is 1.: 1d or 2d and 2.: data, fit or residual.

With these two properties in the both classes of the file, the data from the datacollector can be displayed in a good way.

DataCollector.py

As mentioned before, the DataCollector class in the DataCollector.py file handles all the data that is existing at each moment in the program.

To really store the data, a list that contains objects of the type Dataset is introduced in the constructor. In order to fill this list when the DataCollector is told to create a new dataset, an instance of a DatasetCreator from the RandomDatasetCreator.py file is also instantiated.

Now there are two main methods in this class, simulate\_data and update\_dataset. In simulate\_data, all the values from the FitPage in the MainWindow that are needed for actual calculation of a series of datapoints, are collected. Then these values are plugged into the datasetcreator and will return 3 lists with the x values of the calculation, the y values and fitted values. Then, these are returned.

Since the datacollector handles the datasets, a wrapper for calling this simulate\_data method is needed. That happens in update\_dataset. First, we iterate through the existing datasets to check if there is already an existing dataset for the fitpage that we want to calculate a new dataset for. Then, if there is no existing dataset (existing\_dataset\_index == -1), are completely new dataset can be calculated and appended to the internal list. Note that the plotpage\_index is also added to the dataset and this indicates that we can later ask the datacollector, which tab or page of the plotwidget the data is currently displayed in.

Else, if the dataset is already existing, the x, y and y\_fit data and the property that checks if this dataset is 2d need to be updated. That is done in the next part by creating some random data again and then exchanging values for the existing dataset in the list.

In order to find existing datasets in the datacollector, one can search for a dataset by either the fitpage\_index or by the data\_id. If we know the fitpage index of a dataset and can therefore locate it in the datacollector, we can get the properties of this dataset by the other methods.

Last, the set\_plot\_index method will be used when real plotting is happening in the plotwidget to write this information to the dataset again.

Dataset.py

The dataset contains the lists of numbers from the calculation in this program. The most unique thing in the dataset is the generation of a data\_id that will be determined on the current time and the fitpage before that.

With the other methods, they are mostly getters and setters for the data that is stored in a dataset.

DataTreeWidget.py

Both of the DataTreeWidgets are extending the normal QTreeWidget to change their behavior towards drag and drop.

In the DataTreeWidget class, the constructor is only setting the geometry, enabling drag, set the amount of columns and add the header name for the first and only column.

In the only overridden method from the parent startDrag we allow an item to be dragged, if it is an instance of the earlier discussed data item (contains a link to data or fit from the datacollector)

The drag and drop process right now is realized by working together with a serialized pointer to the object that you are dragging. The pointer can then be deserialized on dropping and then the properties of the original object can be loaded. Since there is no drop functionality involved in the DataTreeWidget itself, this only play a role in the PlotTreeWidget.

PlotTreeWidget.py

The starting of a drag in the PlotTreeWidget is only allowed for modifiers. These can be dragged onto plottables and manipulate them in that way. These modifiers are then considered upon plotting and can edit the line color, the line style for 1d or the color scheme for 2d plots.

The dropping on the other hand is allowed for items that come from the DataTreeWidget. In retrospective it might have been easier to realize both applications in one TreeWidget, since that might be easier for drag and drop operations. With the current implementation, the dragMoveEvent checks if the drag object has an “ID” stored in it that was given in the start of the drag. If it does, we can nearly be sure that this is a data object coming from the DataTreeWidget. Then we accept the drop if the targeted item is a plot item. Otherwise it is ignored. The same is done for the modifier, but here the targets needs to be a plotitem or a plottable item. This needs change, since 1d modifiers should normally not be draggable onto a whole plot.

The drop mechanism for the modifier works with a bit of serialization of pointers, since that was the easiest way for me to achieve real dragging and dropping of an object that is a subclass of a Qt class without losing the information of the subclass.

For the DataItem it is easier, because the data\_id and the type\_num can simply be serialized into a string and can then be processed in the byte array that will be given to the PlotTreeWidget.

The dropping is also split, if the dropping is accepted, will be decided in the dragMoveEvent by event.acceptProposedAction, where the proposed action is a drop.

In dragMoveEvent, we check if the item from the drop is a DataItem (it has an “id”) and accept afterwards. For the Modifier we check if the drop has a modifier “channel” and accept for both targets, the plot and the plottable.

In dropEvent the last step will always be to emit a drop signal and to tell other widgets what functions now need to be executed, in our case this is of course to redraw a tab, because the data that we want to display has changed or a modifier has been added onto an existing plottable or plot. From the beginning of dropEvent, we start with checking again if the drop is a DataItem when it has an id. If this is true, we stream the data\_id and the type\_num from the strings that were in the QByteArrays of the mimeData. If the target for this data, that should be displayed, is a PlotItem, we can accept, create the new child of the targetItem and then emit the event so that the redrawing for the associated tab can start.

As mentioned earlier, the drop for the modifier is a little more complicated, because I directly wanted to use the class structure that was invented for checking, which modifier is which. (color, linestyle, colorscheme) This is done by putting the memory address of the modifier into the QByteArray in the mimeData in the drop and then deserializing this afterwards and add the new modifier by using the information of the original modifier. For the pointer operation, we need the ctypes package here.

PlotWidget.py

The file for the PlotWidget class is not very big. The constructor contains links to the dataviewer and the datacollector to be able to access the data and the treewidgets when actually plotting in a subtab. The PlotWidget is a subclass of the Qt class QTabWidget and therefore manages the outer or major tabs.

The PlotWidget contains two methods to interact with information that will later lay in the widget in each tab, get\_subtabs and get\_figures. As the names indicate, the fitpage\_index can be given and then either the SubTabs item itself or the list of figures from this SubTab is returned.

The most important method here is redrawTab, which extends the redraw method from the DataViewer before. It checks, if the tab for the fitpage already exists. If it doesn’t, it knows that there is a new Tab that needs to be added to itself. Otherwise, the existing plot for this fitpage will be removed and added at the same index again afterwards. This updates the plots in this new tab be recreation.

The most important part in this method is the creation of the SubTabs object that will be stored in the tab of the plotwidget itself.

SubTabs.py

SubTabs.py is the method that does the plotting itself and the managing of the minor tabs or subtabs.

The first class in the file is an extension of the FigureCanvas for Qt that we are using to present a matploblib canvas later in a Qt Widget. To make the figure interactive, the onclick method provides a system that is able to exchange the position of two plots on click.

The second class is the SubTabs itself. This provides the first function grayOutOnDock, that tells the dock widget, that the mainwindow that it is container that it can change its color to gray if it sees, that the current window is docked out.

After that there is the constructor, which is called when the Tab is added to the PlotWidget and we can expect, that we now need to plot something. Therefore, we provide a link to the datacollector to be able to access the data and create and empty list where later the figures of all subtabs can be saved in.

The main stage of plotting is iterative, since we also built up our PlotTreeWidget iteratively and representative of what needs to be displayed in the subtabs and the figures. Because of that, the constructor of subtabs needs a TabItem, so that it can iterate through its children and then knows, how many subtabs need to be created and how many figures have to be in each of them.

The final SubTab with everything that has to be created first is only added to the widget in the end of the method, but at the beginning, the figure and the canvas for the figure are created and also a matplotlib toolbar is integrated into the canvas with respect to the figure.

Then it is decided how many plots need to be in the subtab that is currently created and therefore the children of the current child of the TabItem can be counted. The next bit of code then fixes the plot structure that is displayed in the figure. If you had 3 plots that need to be shown for example, 1 of them would be displayed in a larger tile on the right side, while 2 of them are smaller and in a column on the right side. That is what these gridspecs are used for in the case where the subplot\_count is bigger than 1.

Then the real plotting can begin and first, the axes title is set by the name of the plot item. When iterating over the plottable items for a plot afterwards, we need to know, if an item is a modifier or a plottable, since only the plottable can be plotted. Then, if it is a plottable and it is 2d, the children of the plottable are checked for a colormap and also the data from the datacollector is collected. The type num after that decides, if the data, the fit or the residuals are plotted and also the residuals need to be calculated first.

This process is pretty similar for the 1d case. Here, we set the y axis as a log scale in case of data and fit but as linear in case of residuals and plot it. The plottables children are also iterated over afterwards and then the line modifiers are added to the last plotted line.

Then, only the containerization needs to be applied, so that the docking function is enabled. An overview of this structure is given by this flowchart:

*As a flowchart (?): QTabWidget->QMainWindow->QDockWidget->QWidget->QLayout of former QWidget->FigureCanvasQtAgg*

This is also in the integration branch in GitHub.

There will also be a separate discussion on how far the integration progress is in the integration branch.

Regarding the .txt files in this Branch:

In all the txt files there is some general thoughts on the demo and the integration, but Agenda.txt is definitely the most structured one and could be a good addition to this when reading.

The plotting refactoring\*\*.txt are files that I wrote during some of the meetings we had when we discussed more or less concrete ideas for the demo.

The SasView plotting sketch was something that I made up in the beginning, when I could not directly visualize, how this works in SasView. The list of steps in the upper left corner should be pretty accurate.

The following is regarding the state of the integration of the demo into SasView.

My first idea was to integrate something useful and visible, so I tried the following approach:

The GuiManager owns a new widget, the TabbedPlotWidget that will show all the generated plots from the standalone windows in its own structure. Therefore, I added the new file TabbedPlotWidget with a class inside with the same name. This widget gets initialized in addWidgets() in the GuiManager during its construction when the SasView application is starting. The initialization of the GuiManager is directly invoked through the MainWindow.

Then my idea was to go to the spots in the code where the standalone plots are created and redirect a copy of the axes from a window to the TabbedPlotWidget, so that it could display it aswell. This did not work very well, because to my experience, axes cannot be copied throughout different figures in matplotlib.

Since that did not work, I found another work around. Right now I create the axes that are needed to display something in the figures in the TabbedPlotWidget while they are needed in the code where the plotting for the standalone windows is also taking place. After that, in the PlotterWidget and PlotterBase the axes that we gave in (the one that will be displayed in the TabbedPlotWidget) will be populated with the same stuff that is also populating the standalone plots. This is done in Plotter.py in lines 34f. and 144f.. Then in lines 149f., the for loop iterating over both axes indicates, that the same stuff is done with them, for example plotting, changing titles, changing labels, etc..

The other main additions to the code are in SubTabs.py. The scheme of adding a dock widget to the tab structure is realized in there and also the part where clicking on a figure makes plots change positions. These parts in the code are commented and should need no further mentioning here.