IMASViz Documentation

Release 2019

Oct 28, 2019

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Description

IMASViz is a post-processing tool for visualization and data analysis of the static and dynamic data, stored within **Integrated Modelling Analysis Suite** (**IMAS**) database pulse files e.g. **Interface Data Structures** (**IDSs**). To perform those features it provides a convenient Graphical User Interface (GUI) that allows easier and more straightforward handling of the available signals. The main IMASViz features are done through plotting the data, representing it in a form of plots/graphs. While the tool itself is already available for use it is still under active development, and various features, GUI improvements etc. are still being implemented.

For additional support or if any issues are found with IMASViz please contact the developers via e-mail or submit a ticket on JIRA ITER Webpage.

While submitting the ticket please use the options listed below

Field	Required Option
Project	IMAS (IMAS)
Components	VIZ

Developers:

- **Ludovic Fleury** (CEA Cadarache, Research Institute for Magnetics fusion, e-mail: Ludovic.FLEURY@cea.fr)
- **Dejan Penko** (University of Ljubljana, Mech.Eng., LECAD Lab, e-mail: dejan.penko@lecad.fs.uni-lj.si)

The tool uses the following Python packages:

1. PyQt5

PyQt5 is a comprehensive set of Python bindings for Qt v5. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms including iOS and Android.

Qt is set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems.

For more on **PyQt5** see PyQt5 webpage. For more on **Qt** see Qt webpage.

2. pyqtgraph

Pyqtgraph is a graphics and user interface library for Python that provides functionality commonly required in engineering and science applications. Its primary goals are a) to provide fast, interactive graphics for displaying data (plots, video, etc.) and b) to provide tools to aid in rapid application development (for example, property trees such as used in Qt Designer).

PyQtgraph makes heavy use of the Qt GUI platform (via PyQt or PySide) for its high-performance graphics and numpy for heavy number crunching. In particular, pyqtgraph uses Qt's GraphicsView framework

which is a highly capable graphics system on its own and brings optimized and simplified primitives to this framework to allow data visualization with minimal effort.

For more on **pyqtgraph** see pyqtgraph webpage.

3. matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

For more on **matplotlib** see matplotlib webpage.

4. sphinx

Sphinx is a tool originally created as a Python documentation generator, but it allows also generating documentation in form of html, latex etc.

For more on **Sphinx** see Sphinx webpage.

IMASViz tool is available on **ITER git repository** (access permission is required) under project **Visualization/VIZ**, branch **viz2.0_develop**.

Direct link to the **IMASViz** git.iter repository: IMASViz.

1.2 Release notes

1.2.1 Version 2.3.1

Released: To be released

Changes:

- Equilibrium plugin displays prints now requirements in the log
- Equilibrium plugin raises an error if requirements are not satisfied
- Fixing IMASViz menu items of shot views management when using UDA
- Check prerequisites for using UDA
- UDA: removing MAST from available remote machines
- Available UDA remote servers can now be configured from a configuration file
- User preferences available now for colors of nodes containing data and for data selection
- Fixing bug preventing time arrays to be previewed or plotted
- Update of the README file

1.2.2 Version 2.3.0

Released: 18.10.2019

Changes:

- IMAS-2640: Introduced IMASViz variant of Matplotlib exporter (overwrite the faulty pyqtgraph default Matplotlib exporter).
- Add Makefile for generating the IDSDef_Parser.py files instead of keeping them in the project GIT repository.
- Improve logging messages.

- IMAS-2629: Enabled creating plots for 0D signals.
- IMAS-2651: Improved the time required to build the tree view.
- IMAS-2641: Added display of sized for 2D signals.
- IMAS-2630: Fixed wrong units.
- Plot Configuration UI improvements:
  - Overall UI improvement
  - Replaced plot line number (marked with #) with colored plot marker.

1.2.3 Version 2.2.5

Released: 3.9.2019

Changes:

- Add support for IMAS versions 3.24.0
- Patches for the generation of IDSDef_XMLParser.py files.
- **Documentation Widget** fix related to 'Contents' component.
- Optimization of the display of the node/signal contents in the **Documentation Widget**.
- Fixed bug when clicking twice on the root node resulted in a crash
- Additional checks while plotting added (disabled mixing plots of quantities with different units).
- Added a new command for displaying current selection as IMAS paths.
- Added time unit label for the time slider value in plots as a function of coordinate1.

1.2.4 Version 2.2.4

Released: 1.8.2019

Changes:

- Minor code improvements and fixes.

1.2.5 Version 2.2.3

Released: 30.7.2019

Changes:

- Improve customization of legend labels in the plot configuration UI.
- IMAS-2475: Fixed display of multi-line strings (e.g. ids_properties.comment).

1.2.6 Version 2.2.2

Released: 5.7.2019

Changes:

- Add support for IMAS versions 3.23.3

- Improved data handling and checks for the signal paths and occurrences.

1.2.7 Versions 2.1.0 - 2.2.1

Released: 2.7.2019

Changes:

- Add support for IMAS versions 3.22.0, 3.23.1, 3.23.2
- Improvements for the features: - Export IDS, - 1D plotting, - UDA, - plot legend labels (in case when using UDA)
- Introduce development of standalone UI plugins (using QtDesigner) in a way that they can be also embedded within IMASViz (HowTo documentation included)
- Addition of SOLPS plugin (suitable for reading Edge Profiles IDSs written by SOLPS-ITER)
- Patch for handling Core Profiles IDS profiled_1d array
- Work done tickets:
  - IMAS-2387: Changed string on IMASviz display from 'IMAS database name' to 'TOKAMAK'.
  - IMAS-2404: Highlight/Enable only populated IDSs in the IMAS tree.

1.2.8 Version 2.0.0

Released: 4.2.2019

Changes:

- **Full GUI migration from wxPython and wxmPlot to PyQt and pyqtgraph Python libraries** (including Equilibrium overview plugin)
- Basic plot feature performance improved greatly. Quick comparison for plotting 17 plots to a single panel using default plotting options: - wxPython IMASViz: ~13s - PyQt5 IMASViz: less than 1s (more than **13x speed improvement**!)
- Improved tree view build performance (wxPython IMASViz was practically unable to build tree view for arrays containing 1500+ time slices)
- Superior plot export possibilities
- GUI improvements
- Database tree browser interface display improvements
- Added first 'node contents display' feature (displayed in the *Node Documentation* Widget)
- Reduced the number of separate windows, introduce docked widgets
- Introduce first GUI icons
- MultiPlot feature relabeled to TablePlotView
- SubPlot feature relabeled to StackedPlotView
- Add support for IMAS versions 3.19.0, 3.20.0, 3.21.0 and 3.21.1
- Included **documentation + manual** (~60 pages in PDF) in a form of reStructuredText source files for document generation (single source can be generated into multiple formats e.g. PDF, HMTL. . .)
- In-code documentation greatly improved and extended

- and more…

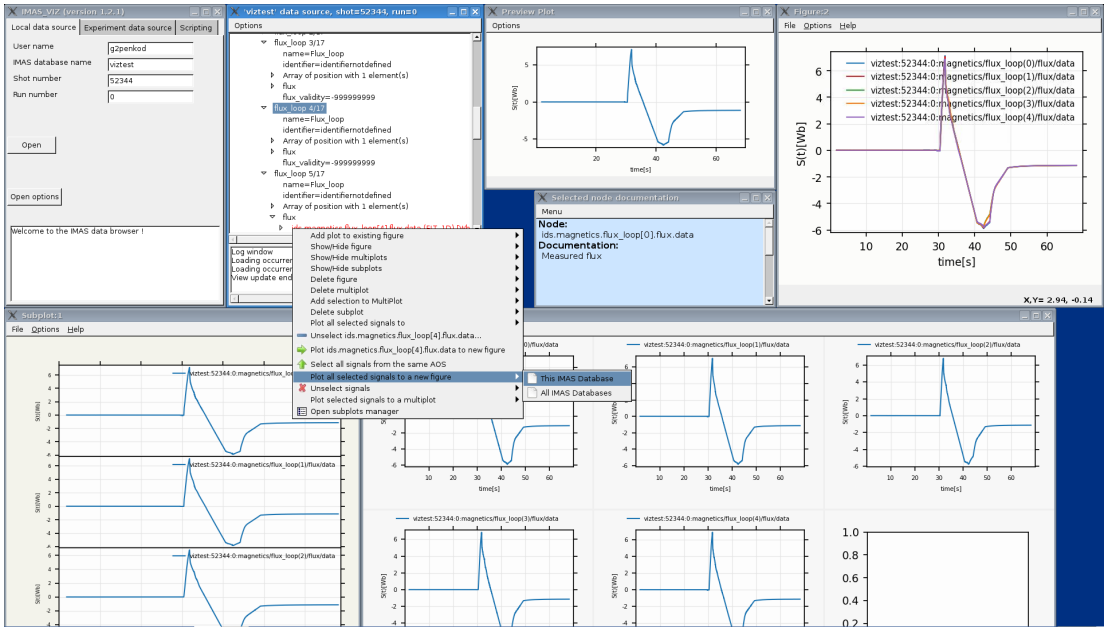
Short summary of files and line changes count (ignoring generated files and scripts):

- 193 commits,
- 268 files changed,
- 13316 insertions(+),
- 10162 deletions(-)

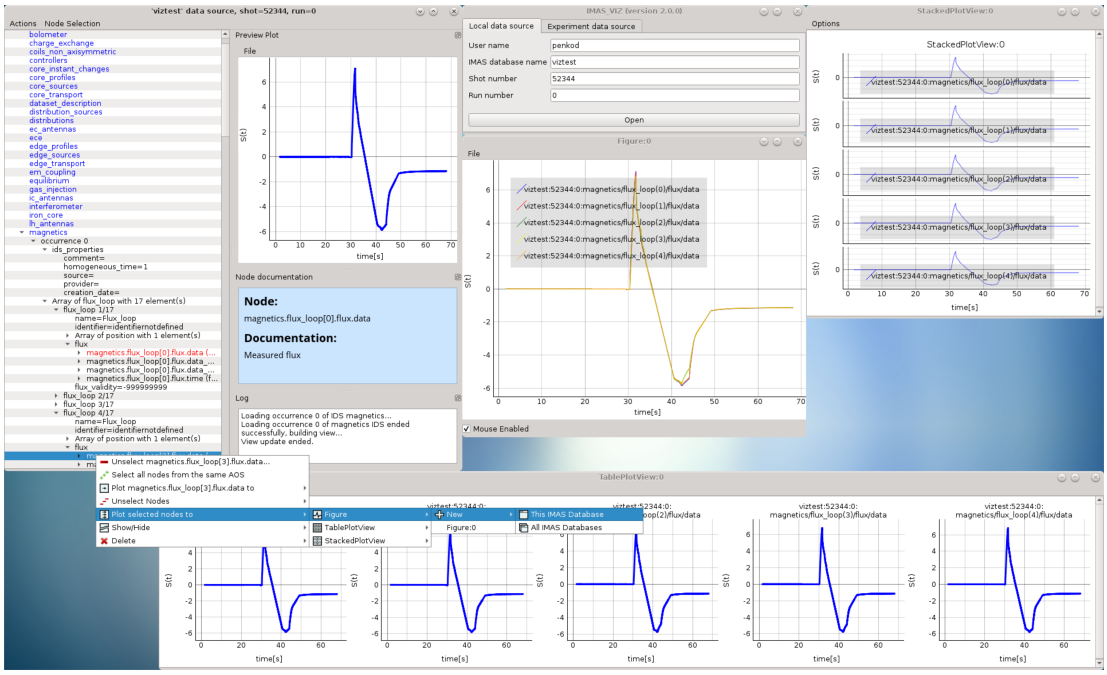
Note: The migration to PyQt5 due to IMASViz containing a large code sets is not yet fully complete. List of known features yet to migrate to IMASViz 2.0: `Add selected nodes to existing TablePlotView`, and `StackedPlotView manager`.

A quick GUI comparison between the **previous** and the **new** IMASViz GUI is shown below.

Overview of IMASViz 1.2 GUI:



Overview of IMASViz 2.0 GUI:



1.2.9 Version 1.2

Released: 24.8.2018

Changes:

- New functionality: selection command of nodes belonging to same parent AOS (Array of Structures)
- MultiPlot and SubPlot design improvements
- Adding support for IMAS versions 3.19.0

1.2.10 Version 1.1

Released: 8.6.2018

Changes (since March 2017):

- Bugs fixes & performance improvement
- Code migration to Python3
- GUI improvements
- UDA support for visualizing remote shots data
- Reuse of plots layout (multiplots customization can be saved as a script file to be applied for any shot)
- A first plugins mechanism has been developed which allows developers to integrate their plugins to IMASViz
- The 'Equilibrium overview plugin' developed by Morales Jorge has been integrated into IMASViz
- Concerning UDA, WEST shots can be accessed if a SSH tunnel can be established to the remote WEST UDA server.
- Introducing MultiPlot and SubPlot features
- Add support for IMAS version 3.18.0

CHAPTER

TWO

USER MANUAL

Note: The manual presented is executed on the **Gateway HPC**.

2.1 Getting Started

This section describes setting the environment configuration required to run the IMASViz tool and how to run the application itself.

2.1.1 Running IMASViz as a module on The GateWay

The procedure below describes how to use IMASViz if it is available as a module on the HPC cluster.

2.1.1.1 Setting the Environment

In a new terminal, execute the following command in order to load the required modules:

```
module load cineca
module load imasenv   # or any other specific imasenv module version
module load imas-viz/v2

# The next few modules should be loaded together with imasenv module
# Listing them all here in case of module related issues.
# module load itm-gcc/6.1.0
# module unload itm-python/2.7
# module load itm-python/3.6
# module load itm-qt/5.8.0
```

```
# Listing all available imas-viz modules can be done by running
module av imas-viz
```

```
# With the imas-viz module loaded (version 2.3.0 or never)
# the documentation can be accessed by running
biz-doc
```

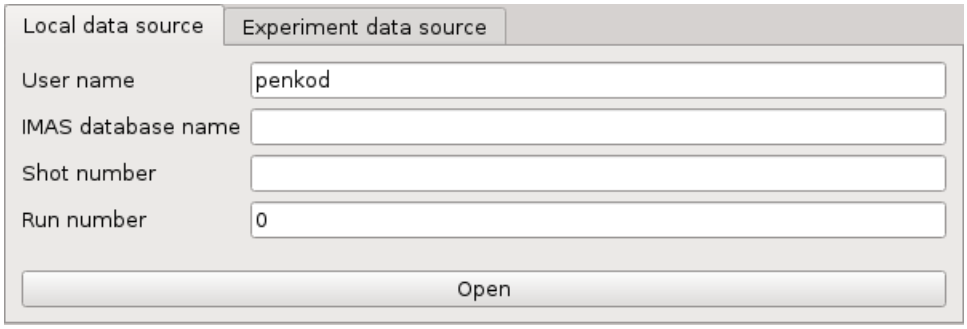
Warning: **IMPORTANT!** IMAS databases (IDSs) were written using specific version of IMAS. In order to open these IDSs the **same IMAS module version** should be used due to possible IDS database structure changes through different versions. Any tools or utilities that work with IDSs, including `IMASViz`, cannot work properly if this "IMAS version mismatch" is too great (!).

2.1.1.2 Running IMASViz

With the environment set, run the IMASviz by simply typing the following command:

```
viz
```

The main GUI window of IMAS_VIZ should display, as shown in the figure below:



The description of the above input parameters is as follows:

GUI Fields	Description
User name	Creator/owner of the IMAS IDSs database
IMAS database name	IMAS database label, usually device/machine name of the IMAS IDS database (i. e. iter, aug, west. . .)
Shot number	Pulse shot number
Run number	Pulse run number

2.1.2 Available benchmark IMAS databases

On the GateWay HPC there are a few **benchmark IMAS IDS cases** available. These databases are the main source of data used for IMASViz testing purposes and were also included in writing the this documentation. Users can freely use them for examples and practice purposes.

Note: There IMAS IDS cases are confirmed to work with IMAS versions **3.19.1** - **3.20.0**.

Available IMAS IDS Case Parameters			
Parameters	Case 1	Case 2	Case 3
User name	g2penkod	g2penkod	g2penkod
IMAS database name	viztest	viztest	viztest
Shot number	52344	52682	53223
Run number	0	0	0

2.1.3 Running IMASViz from source

The procedure below describes how to run IMASViz from source.

2.1.3.1 Requirements

The fundamental requirements in order to locally run IMASViz are:

- IMAS
- **Python3 and Python libraries:**
  - PyQt5
  - pyqtgraph
  - matplotlib
  - Sphinx (**`pip3 install sphinx`**)
  - Sphinx RTD theme (**`pip3 install sphinx_rtd_theme`**)

2.1.3.2 Obtaining the source code

To obtain the IMASViz code source the next two steps are required:

1. Clone repository from **git.iter.org** (permissions are required!).

   Direct link to the **IMASViz** git.iter repository: IMASViz.

2. Switch to IMASViz2.0 branch (required if master branch is not updated yet)

```
git fetch # optional
git branch -r # optional
git checkout develop
```

2.1.3.3 Setting the environment

To set the environment, go to `viz` directory and set *VIZ_HOME* and *VIZ_PRODUCTION* environment variables by running the next commands in the terminal:

```
cd viz
# bash
export VIZ_PRODUCTION=0
export VIZ_HOME=$PWD
# csh
setenv VIZ_PRODUCTION 0
setenv VIZ_HOME $PWD
```

Then proceed with the next instructions.

GateWay HPC

Load next modules:

```
module load cineca
module load imasenv
module load itm-gcc/6.1.0
module load itm-python/3.6
module load itm-qt/5.8.0
```

ITER HPC

Load next module:

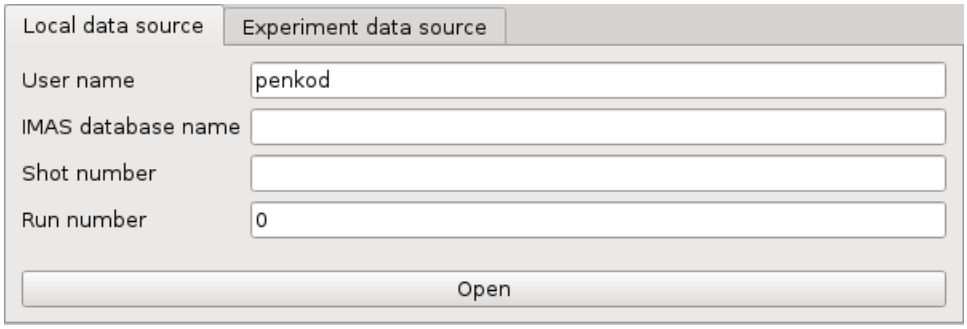
```
module load IMAS/3.20.0-3.8.3
```

2.1.3.4 Running IMASViz

To run IMASViz, run the next commands in terminal:

```
python3 $VIZ_HOME/imasviz/VizGUI/QtVIZ_GUI.py
```

The main GUI window of IMAS_VIZ should display, as shown in the figure below:



The description of the above input parameters is as follows:

GUI Fields	Description
User name	Creator/owner of the IMAS IDSs database
IMAS database name	IMAS database label, usually device/machine name of the IMAS IDS database (i. e. iter, aug, west. . .)
Shot number	Pulse shot number
Run number	Pulse run number

2.1.4 Latest documentation and manual

The documentation provided on other sources (confluence pages etc.) than the project repository might not be up to date. To get the latest documentation, first obtain the IMASViz source code (see *Obtaining the source code*).

Then navigate to

```
cd $VIZ_HOME/doc
```

and run

```
# for PDF documentation
module load texlive
make pdflatex
firefox build/latex/IMASViz.pdf
# for HTML documentation
make html
firefox build/html/index.html
```

Note: Additional prerequisites for generating the documentation: **Sphinx** and **Sphinx RTD** theme (listed in *Requirements*)

2.2 Loading IDS from IMAS local data source

This section describes and demonstrates how to load the **IMAS IDS case** within **IMASViz** and open one of the **IDS nodes**.

Note: The procedure below is executed on the **GateWay HPC** and thus the **IMAS IDS cases** available on the GateWay are used.

2.2.1 Loading IMAS IDS

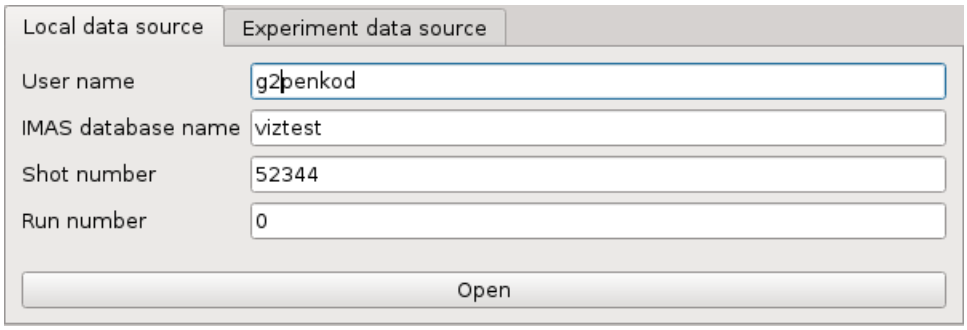
The procedure to load the IDS is as follows:

- In the main *IMASViz GUI*, select the first tab - *Local data source*.
- Enter the following parameters, listed below, to the appropriate text fields.

IMAS IDS case	
Parameters	Values
User name	g2penkod
IMAS database name	viztest
Shot number	52344
Run number	0

By default, the data source is a pulse file located in `$HOME/public/imasdb/<IMAS database name>/3/0/` directory. In this case, the `~public/imasdb/viztest/3/0/` directory of user `g2penkod`.

The filled GUI should then look as shown in the next figure:

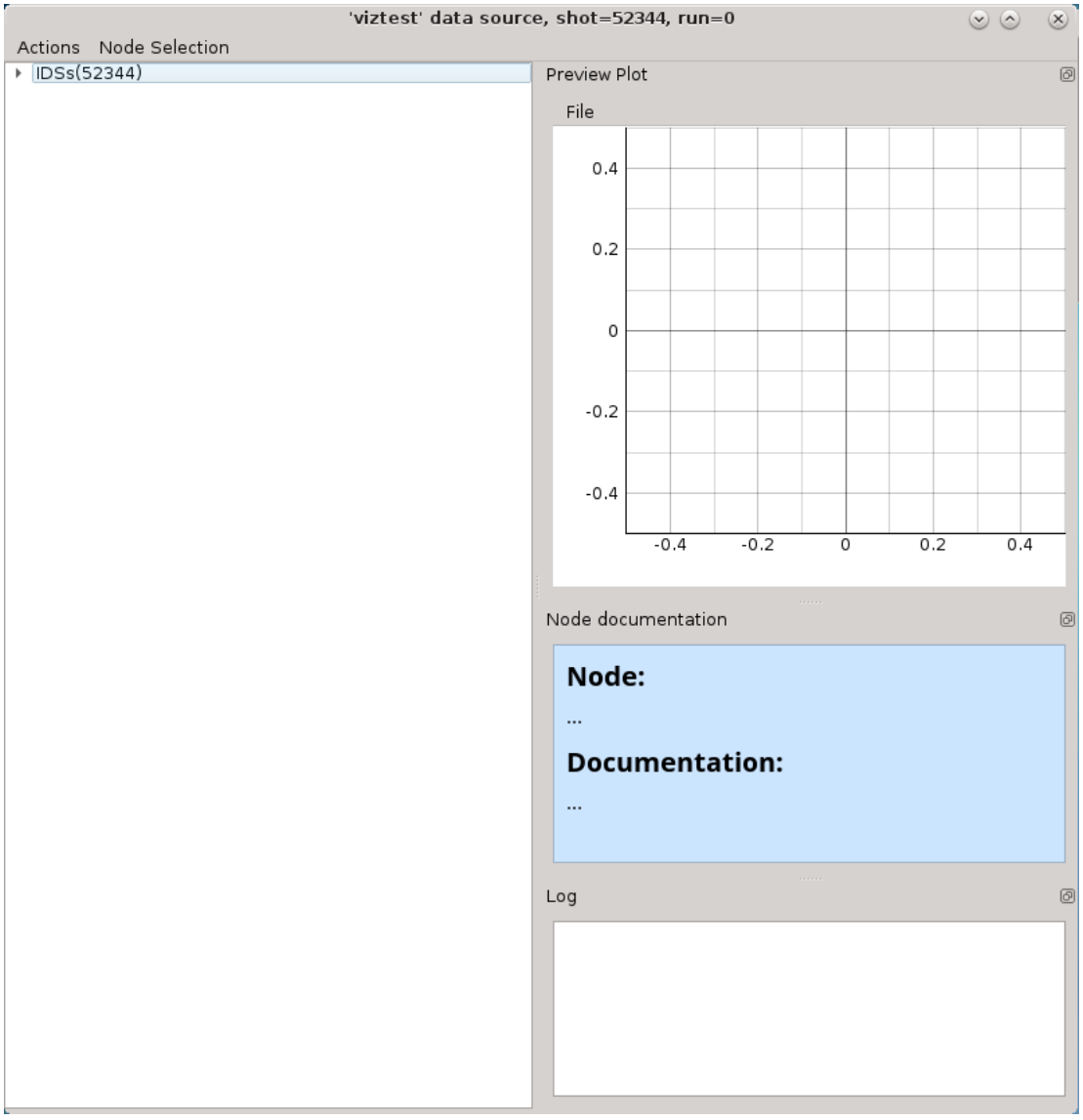


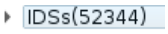
2.2.2 Open IDS

The procedure to open any IDS is the same. In this manual, the procedure will be shown on **magnetics IDS**.

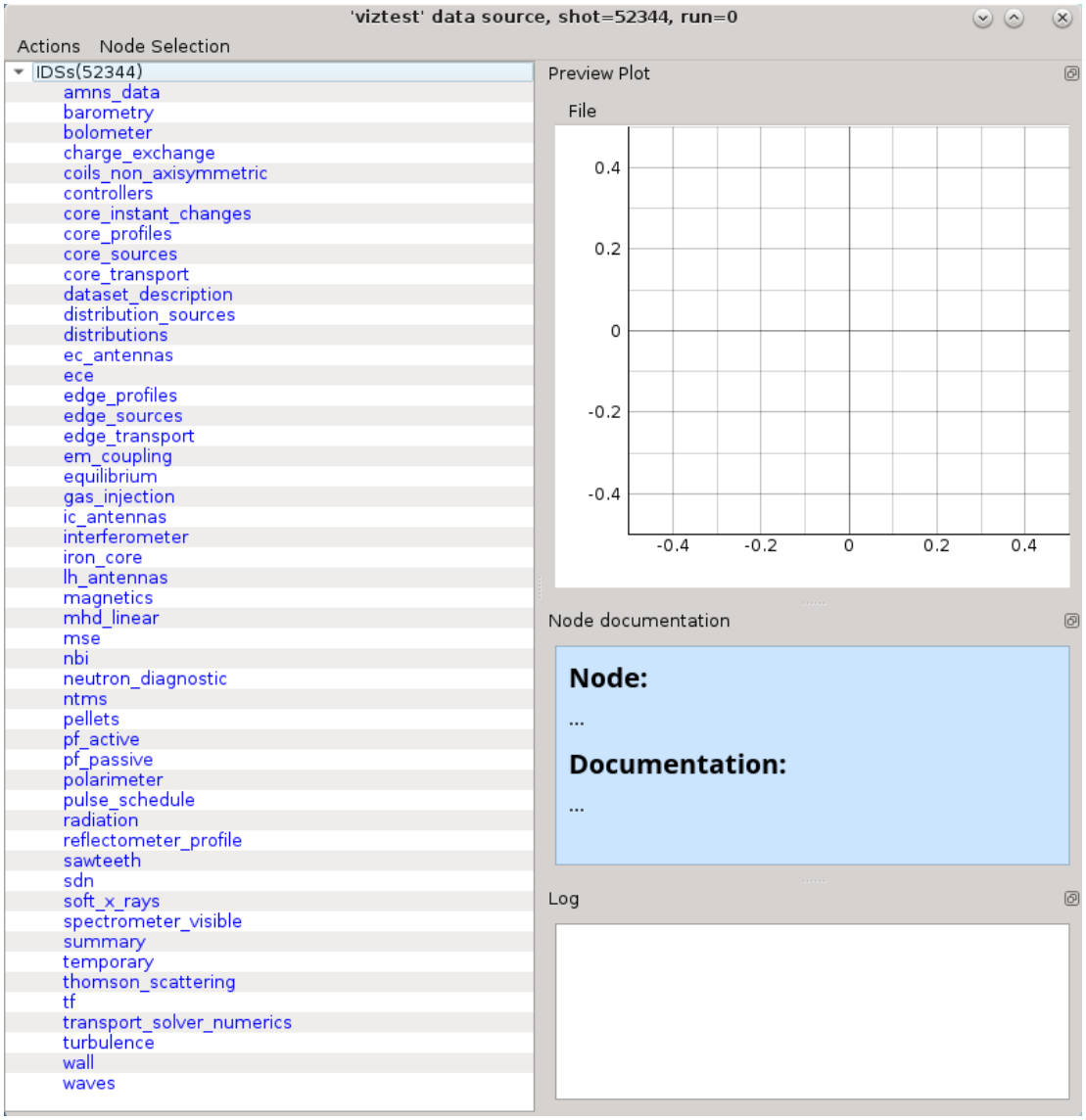
1. Click *Open* button to open the IDS.

   A navigation tree window will open, as shown in the figure below.

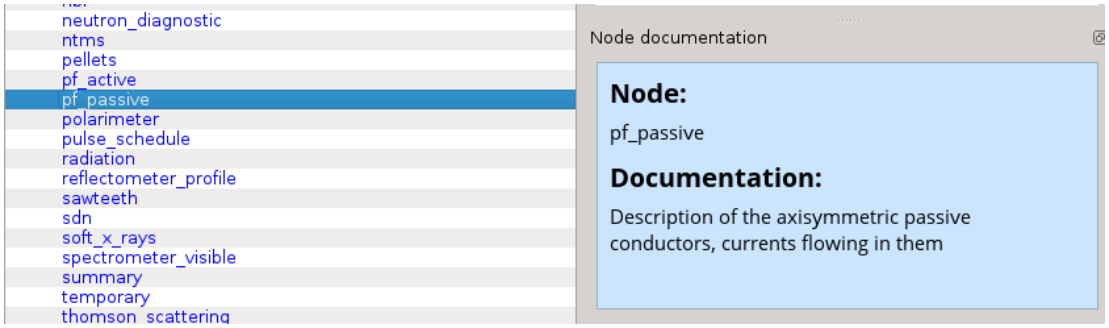


2. Press the **arrow button**  on the left side of the **IDS root node**.

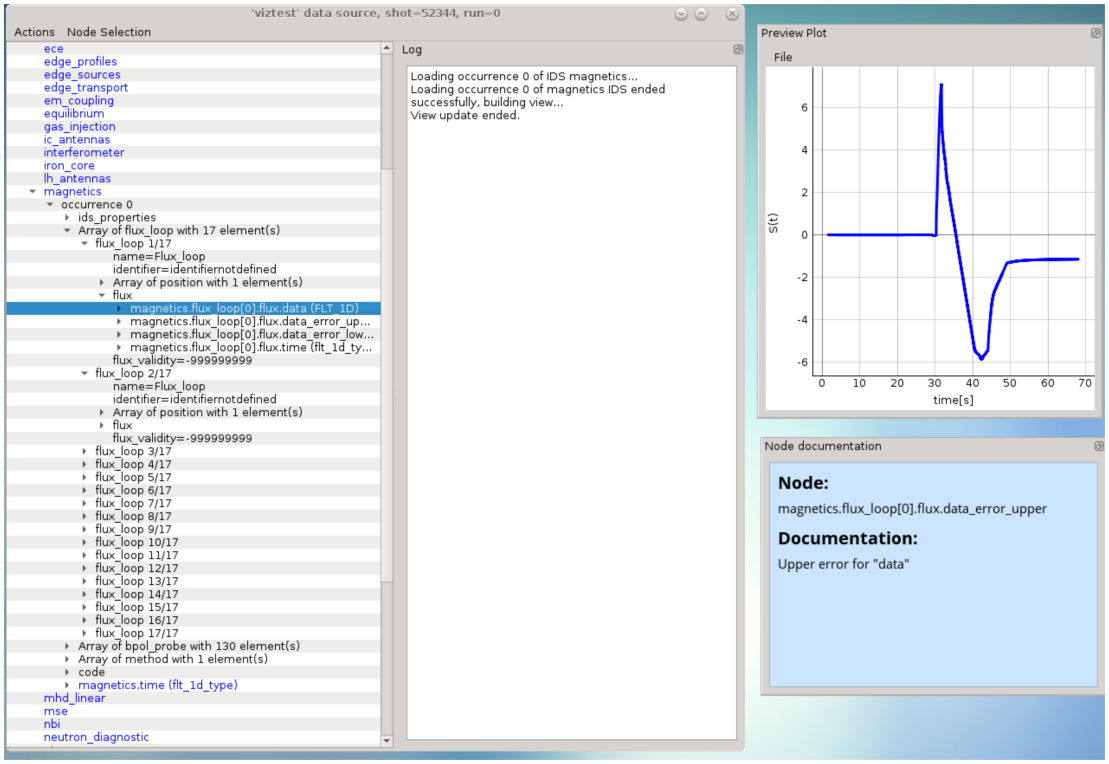
   This will expand the navigation *tree window* and display a list of all IDSs. The tree will allow browsing data for the specific shot number which is displayed by the root node (`IDSs(52344)`).



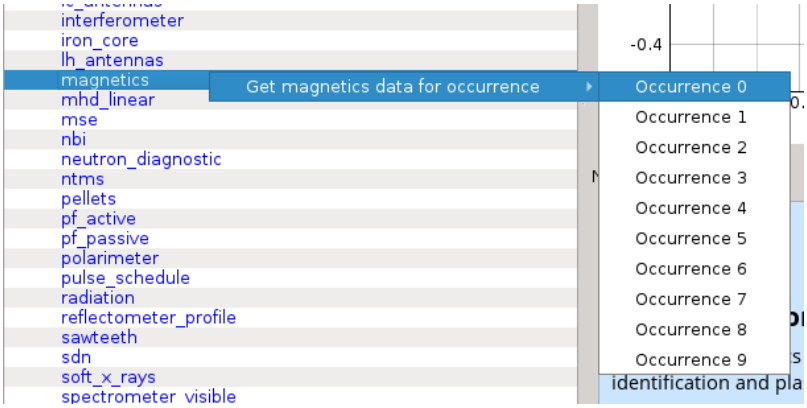
When IDS or node label is selected the *Node documentation* widget will display the basic information (name and documentation) of the node, as shown below.



The *Node Documentation* widget can be freely taken out from the main window by clicking *undock* button  the and positioned anywhere on the screen. The same thing goes for the *Preview Plot* and *Log* widget.

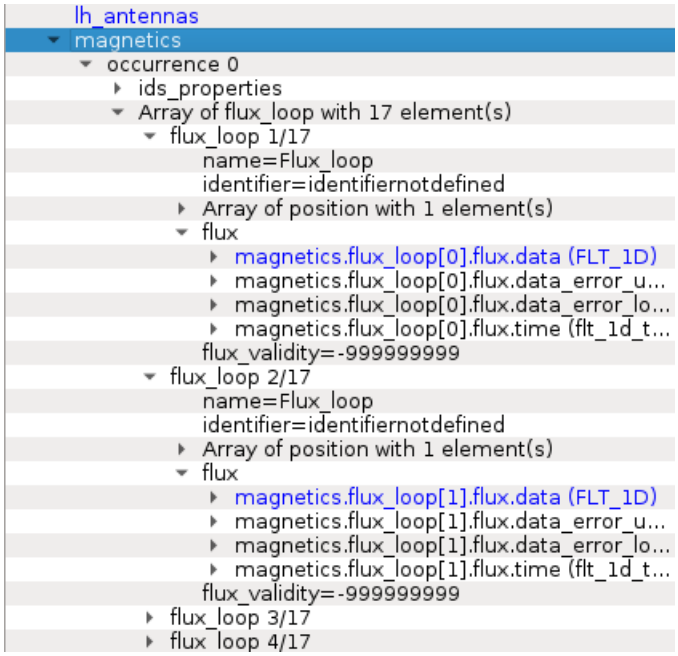


3. Open **magnetics IDS** by right-clicking on the **magnetics** node and selecting the command *Get magnetics data* (occurrence 0) as shown in the figure below.



Note: Alternative: Double-clicking on the **IDS node label** -> **occurrence 0** (default) of the selected IDS will load.

The magnetics IDS nodes are displayed as new nodes in the tree, as shown in the figure below. Nodes of an IDS are organized according to the **IMAS data dictionary**. Inside the **magnetics** tree, plottable **FLT_1D** nodes are colored blue (array length > 0).



2.3 Node selection features

IMASViz offers the user the ability to set or mark a selection of plottable arrays (nodes) as once. This way plotting multiple plots to the same *Figure* or to a *MultiPlot View* is more convenient and faster, avoiding "one-by-one" plotting.

Note: How to plot selection is described later in section *Plotting 1D arrays*.

In the continuation of this section different methods of node selection are described.

2.3.1 Select One-by-one

To select nodes one by one, first, right-click on the wanted node. From the shown pop-up menu, select the command *Select <node name>* ✚.

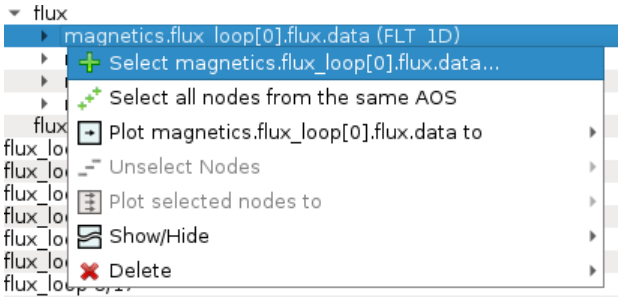


Fig. 1: Selecting a plottable node.

The selected node label gets colored red, indicating that it is added to the selection.

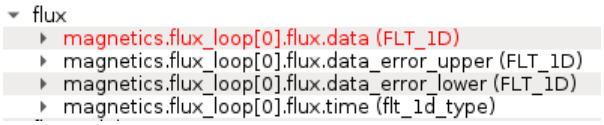


Fig. 2: Node colored red -> node is selected.

Repeat that procedure until all wanted nodes are selected.

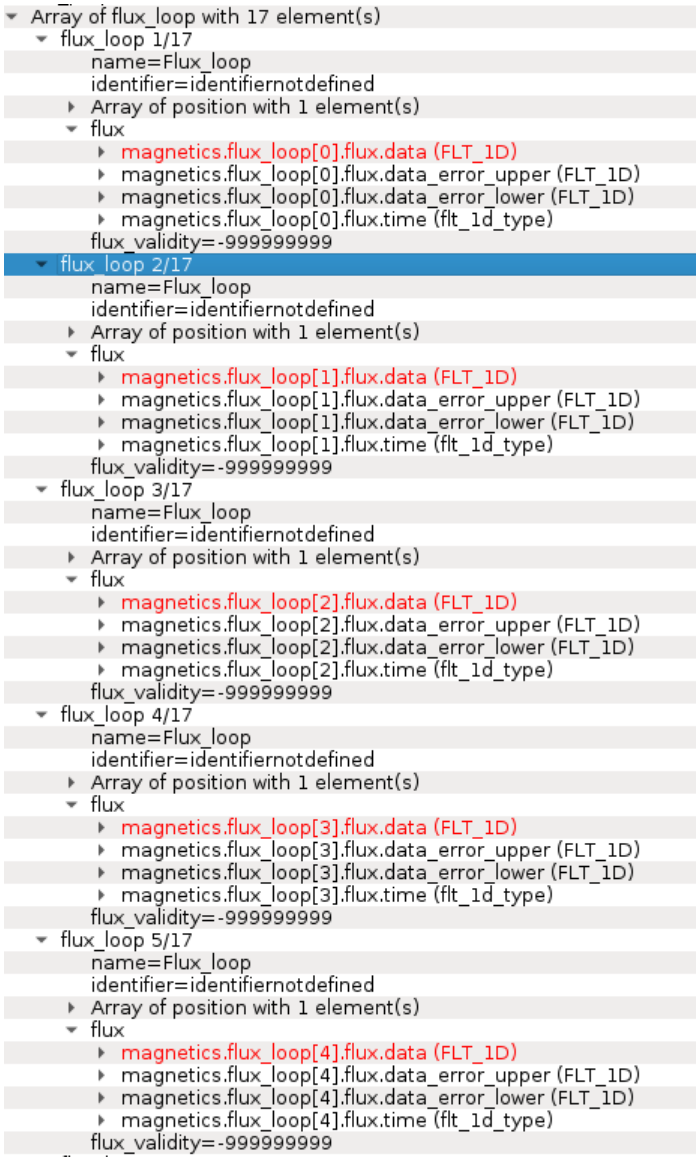


Fig. 3: Example of multiple nodes selection.

Note: At the same time, nodes from other opened IDS databases too can be selected.

2.3.2 Select All Nodes of the same Structure (AOS)

To select all nodes of the same structure (same node structure type), right-click on one of the nodes and from the shown popup-menu select the option *Select All Nodes From The Same AOS* .

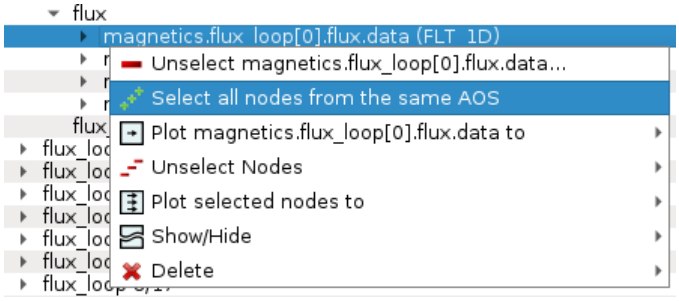


Fig. 4: Selecting plottable nodes of the same structure/type.

All nodes of the same structure will be selected and their label will be colored to red, indicating that they were added to the selection.

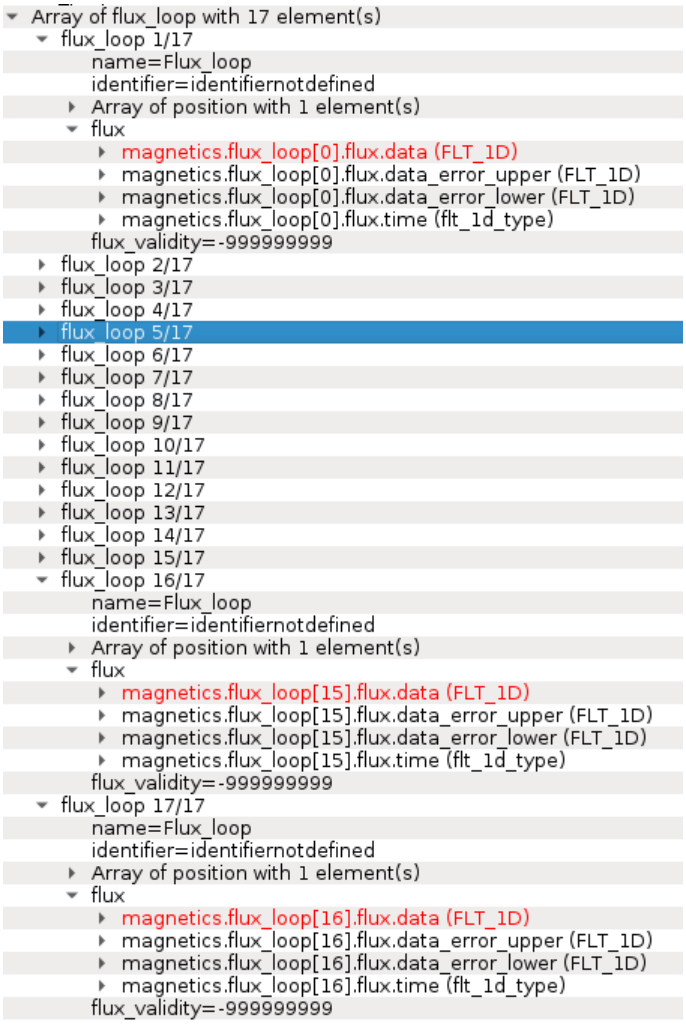


Fig. 5: Node colored red -> node is selected. All plottable nodes of the same structure/type are selected, in this case, 17 nodes.

2.3.3 Save Node Selection Configuration

Any node selection can be saved to a configuration file and used later with any opened IMAS database. To save a selection, follow the next steps:

1. In the main tree browser menu navigate to **Node Selection** -> **Save Node Selection**.
2. In opened GUI window type the name of the configuration.

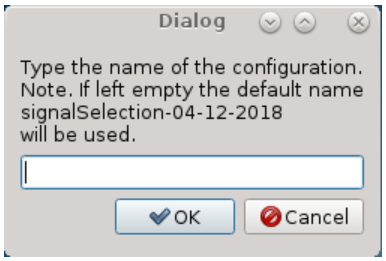


Fig. 6: Save Node Selection Dialog.

3. Press **OK** button.

Note: The configurations are saved to `$HOME/.imasviz` folder.

2.3.4 Apply Selection From Saved Configuration

Applying saved node selection can be performed using both *Node Selection Configuration* and *MultiPlot Configuration*.

2.3.4.1 Apply Selection From Saved Node Selection Configuration

To apply selection from *Node Selection Configuration*, follow the next steps:

1. In the main tree browser menu navigate to **Actions -> Apply Configuration**. In the shown window switch to *Available Node Selection Configurations* tab.

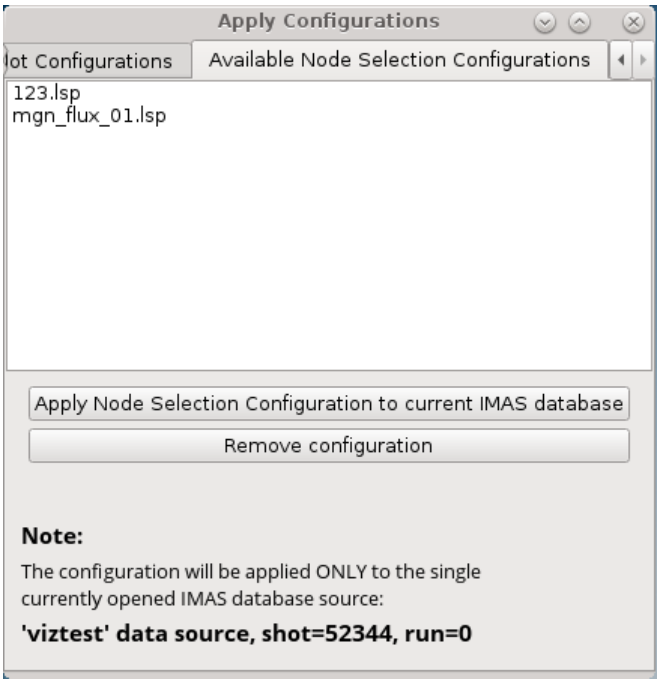


Fig. 7: *Apply Node Selection Configuration* tab.

2. Select the configuration from the list.
3. Press **Apply selection only**.

   The signal nodes, found in the configuration file, will then be selected.

2.3.4.2 Apply Selection From MultiPlot Configuration

To apply selection from *MultiPlot Configuration*, follow the next steps:

See also:

How the create MultiPlot Configuration is described in *Table Plot View*.

1. In Main Tree View Window menu navigate to **Actions -> Apply Configuration**. In the shown window switch to *Apply Plot Configuration* tab.

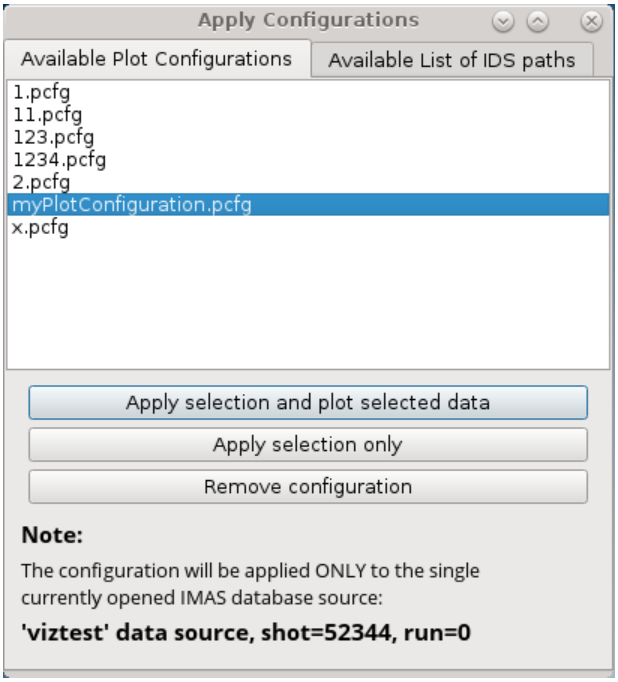


Fig. 8: *Apply Plot Configuration* tab.

2. Select the configuration from the list.
3. Press **Apply selection only**.

   The signal nodes, found in the configuration file, will then be selected.

2.3.5 Unselect selected Node Signals

There are few features that allow node signal unselection.

2.3.5.1 Unselect One-by-one

To unselect nodes one by one, first, right-click on the selected node. From the shown pop-up menu, select the command *Unselect <node name>* .

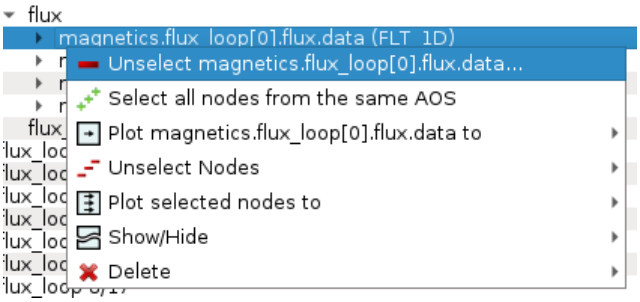


Fig. 9: Unselecting plottable node.

2.3.5.2 Unselect All

To unselect all selected nodes, first, right-click on the selected node. From the shown pop-up menu, select the command *Unselect Nodes* -> *This IMAS Database* or *All IMAS Databases*.

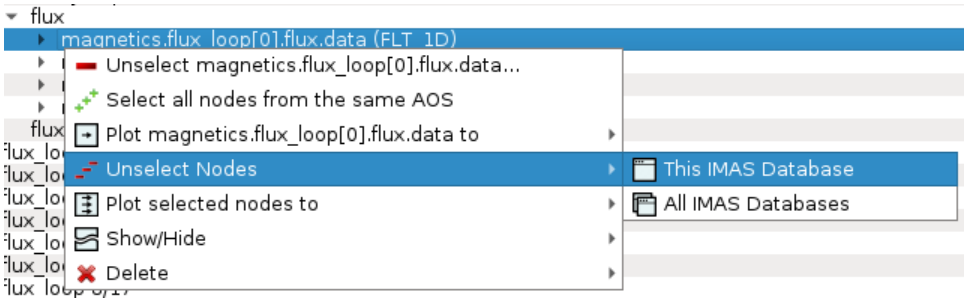


Fig. 10: Unselecting multiple plottable nodes at once.

2.4 Plotting 1D arrays

The plotting of 1D arrays option and plot handling is the main feature and purpose of the *IMASViz*.

This section describes the basics of plotting a 1D array, stored in the IDS, and how to handle the existing plots.

2.4.1 Plotting a single 1D array to plot figure

The procedure to plot 1D array is as follows:

1. Navigate through the **magnetics IDS** and search for the node containing **FLT_1D** data, for example **magnetics.flux_loop[0].flux.data**. Plottable FLT_1D nodes are colored **blue** (array length > 0).

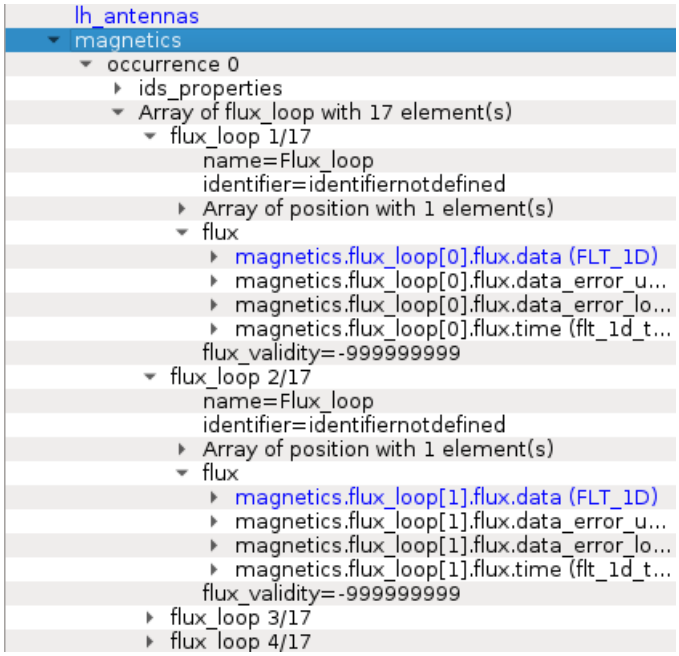


Fig. 11: Example of plottable FLT_1D node.

By clicking on the node the preview plot will be displayed in the *Preview Plot*, located in the main window. This feature helps to quickly check how the data, stored in the FLT_1D, looks when plotted.

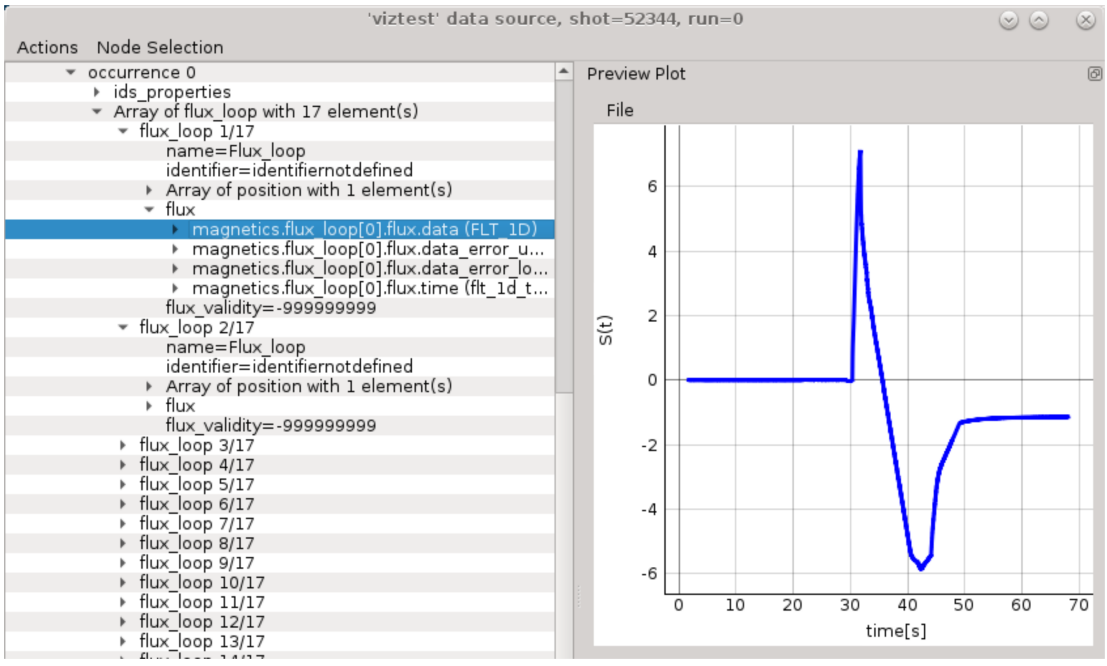


Fig. 12: Preview Plot

2. Right-click on the **magnetics.flux_loop[0].flux.data (FLT_1D)** node.
3. From the pop-up menu, select the command *Plot ids.magnetics.flux_loop[0].flux.data to* -> *figure* -> *New*

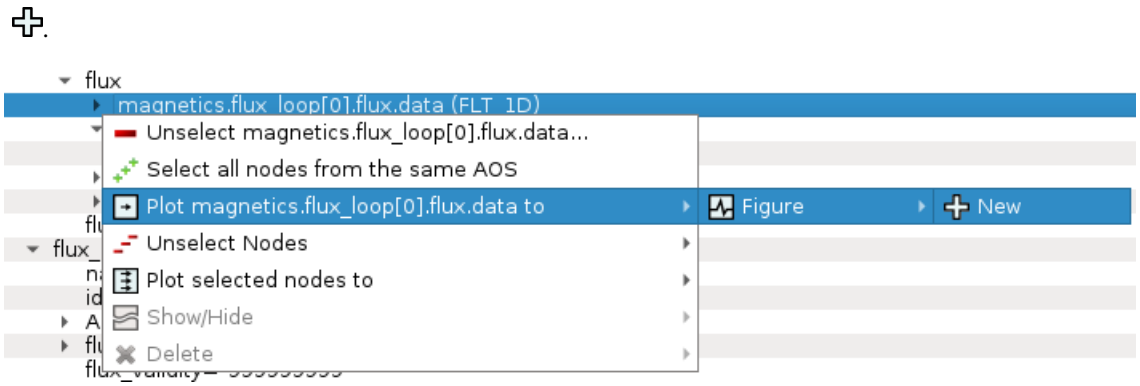


Fig. 13: Navigating through right-click menu to plot data to plot figure.

The plot should display in plot figure as shown in the image below.

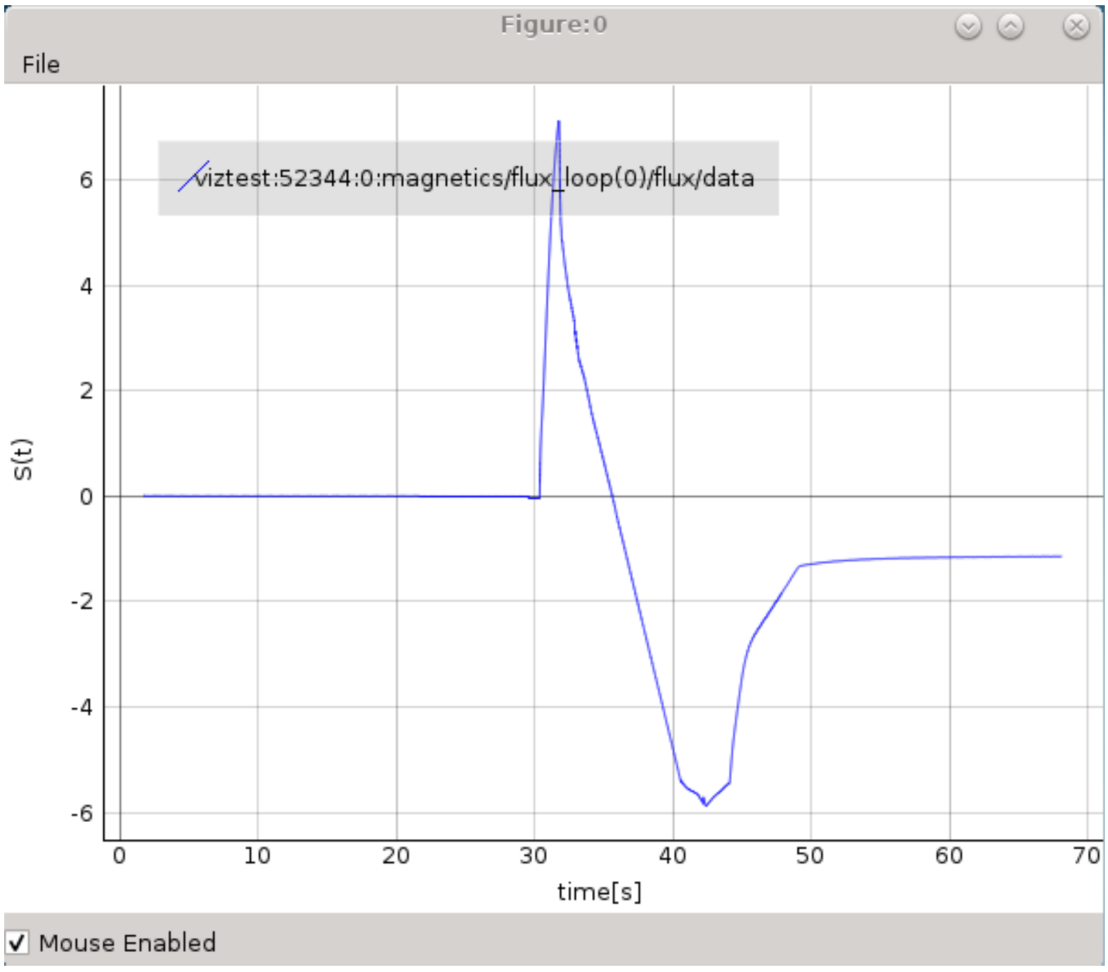


Fig. 14: Basic plot figure display.

2.4.1.1 Basic plot display features

The below features are available for any *plot display*. Most of them are available in the right-click menu.

Note: Term *Plot Display* is used for any base subwindow for displaying plots. Following that the *Plot Figure* contains a single *Plot Display*, while *Table Plot View* and *Stacked Plot View* consist of multiple *Plot Displays*.

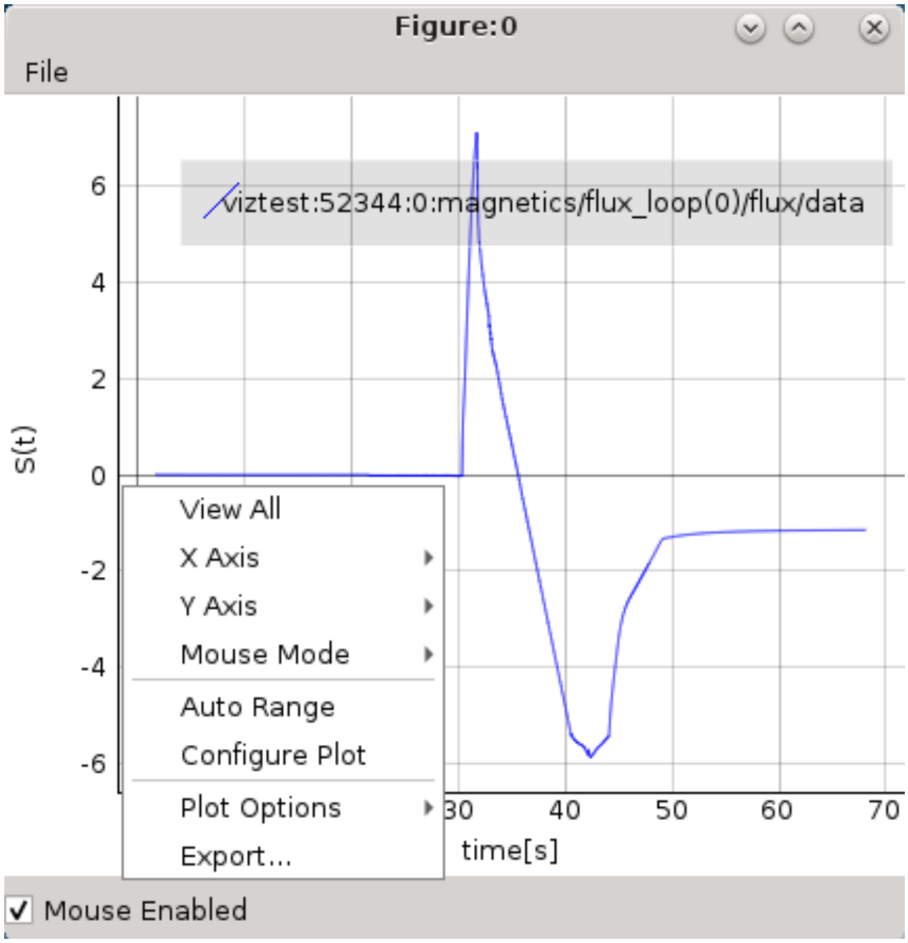


Fig. 15: Plot display window right-click menu.

View All

Zoom to view whole plot area.

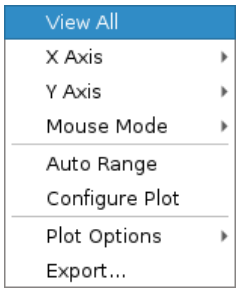


Fig. 16: *View All* feature in the right-click menu.

Auto Range

Similar to *View All* feature with the difference that it shows plot area between values `X_min` -> `X_max` and `Y_min` -> `Y_max`, without additional "plot margins" on the sides.

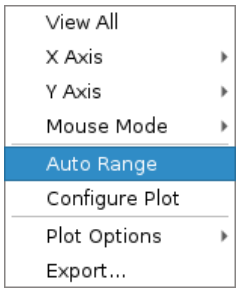


Fig. 17: *Auto Range* feature in the right-click menu.

Left Mouse Button Mode Change

Change between *Pan Mode* (move plot around) and *Area Zoom Mode* (choose selectable area to zoom into).

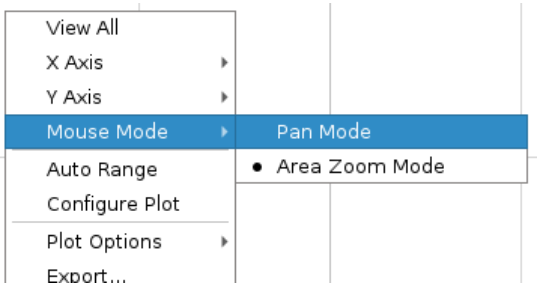


Fig. 18: *Mouse Mode* feature in the right-click menu.

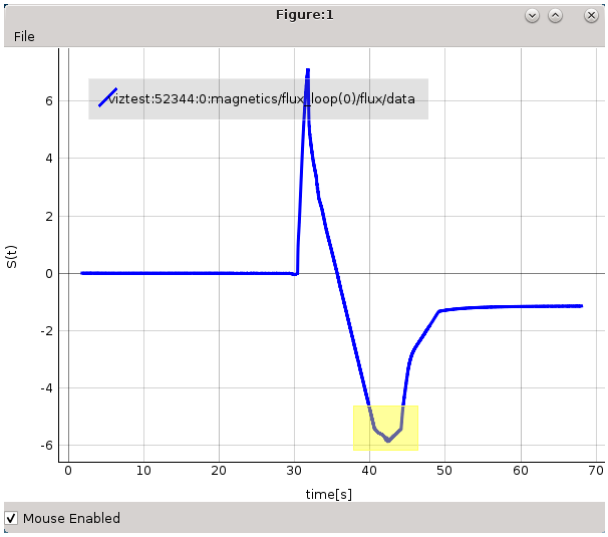


Fig. 19: *Area Zoom* example: Marking zoom area using.

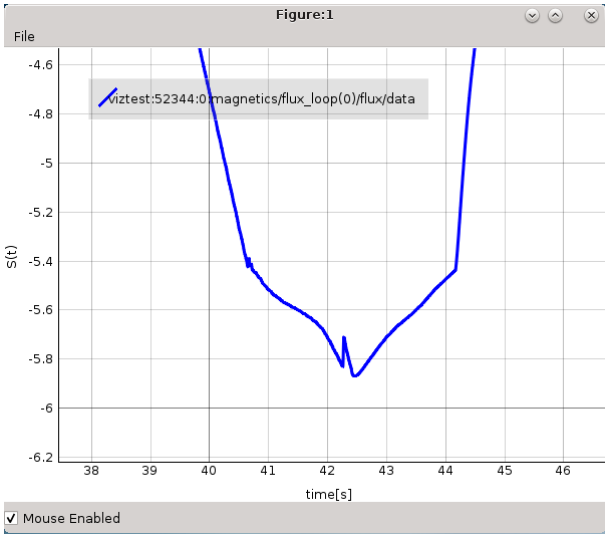


Fig. 20: *Area Zoom* example: Result.

Axis options

X and Y axis range, inverse, mouse enable/disable options and more.

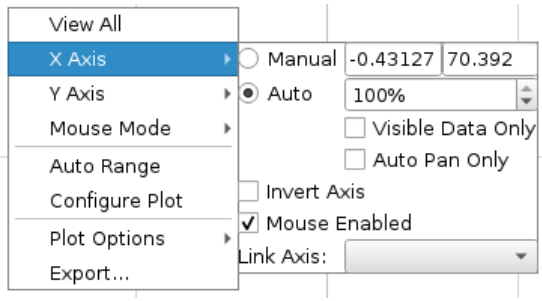


Fig. 21: *Axis Options* feature in the right-click menu.

Plot Configuration and Customization

Setting color and line properties of plots shown in the Plot Display.

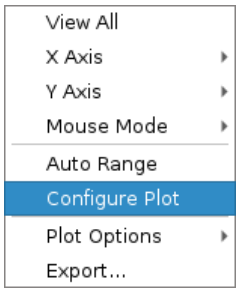


Fig. 22: *Configure Plot* feature in the right-click menu.

Each plot can be customized. By selecting this feature a separate GUI window will open, listing all plots within the plot display window and their properties that can be customized.

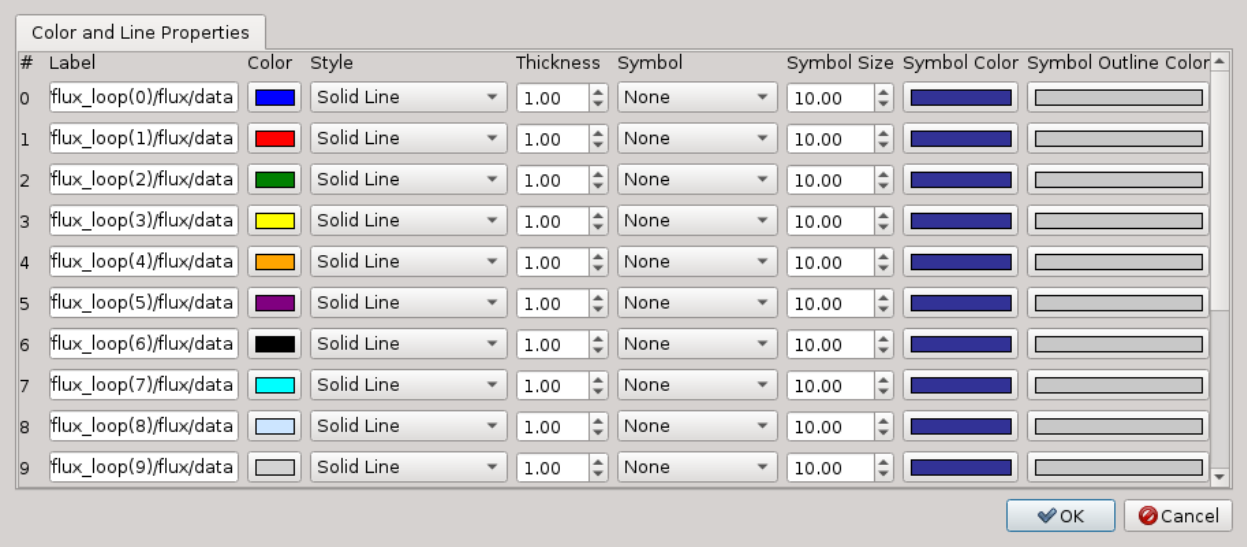


Fig. 23: *Configure Plot* GUI.

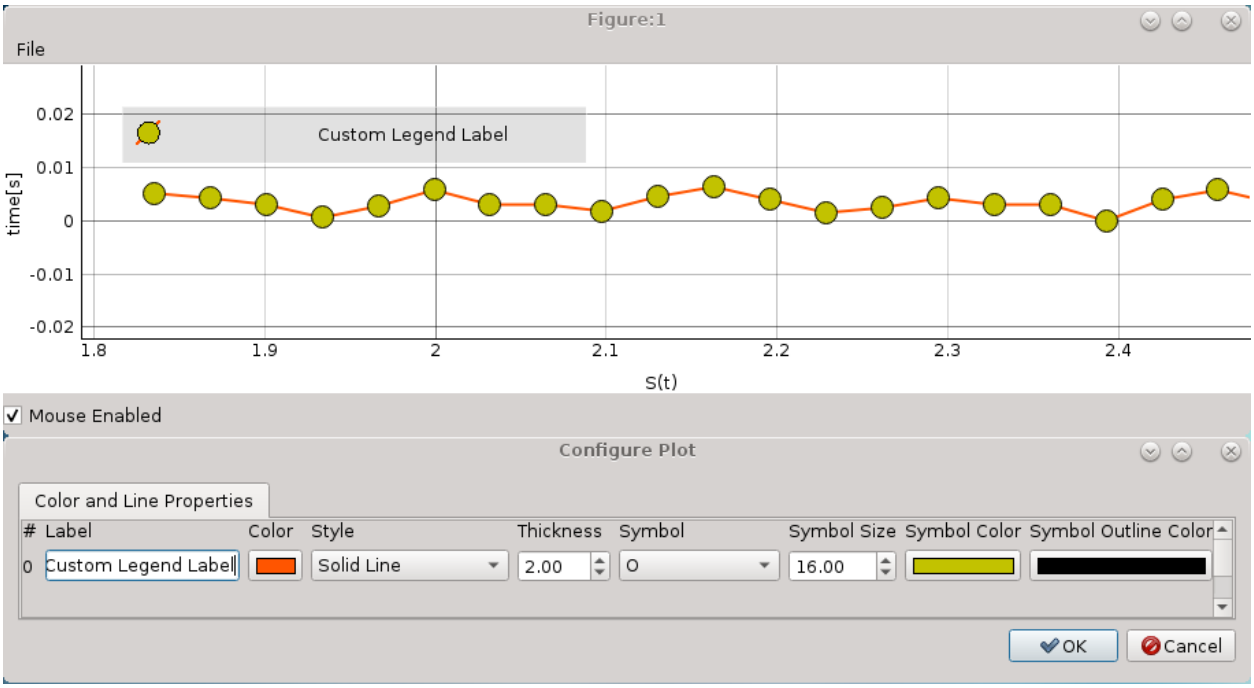


Fig. 24: Plot configuration example for single plot.

Plot options

Enable/Disable grid, log scale and more.

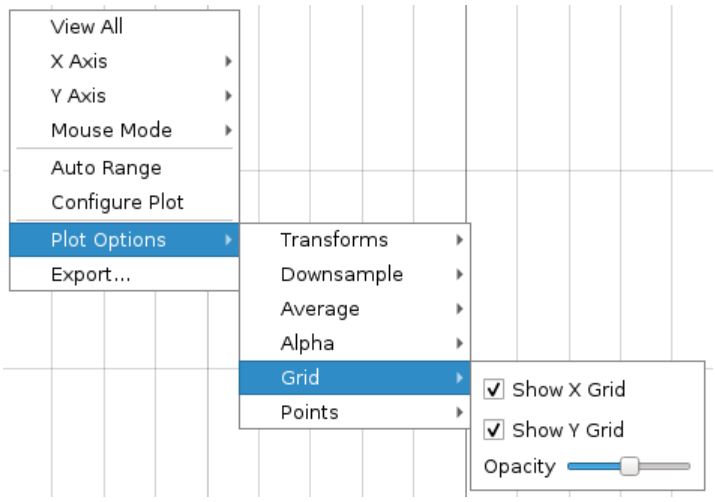


Fig. 25: *Plot Options* feature in the right-click menu.

Export feature

The *Plot Display* scene can be exported to:

- image file (PNG, JPG, . . .). A total of **16** image formats are supported.
- scalable vector graphics (SVG) file
- matplotlib window
- CSV file
- HDF5 file

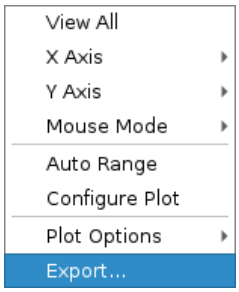


Fig. 26: *Export* feature in the right-click menu.

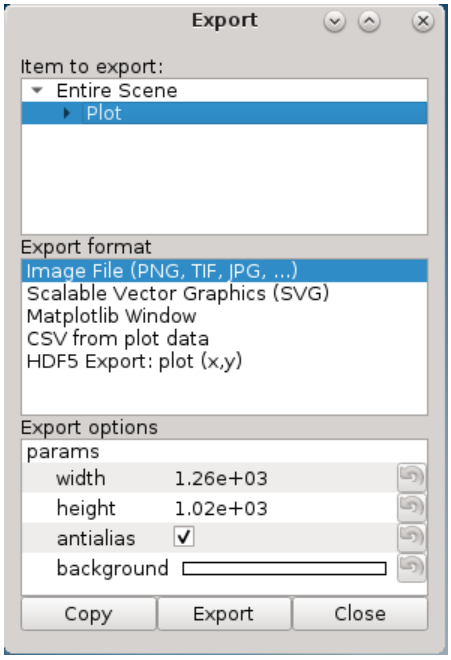


Fig. 27: Export GUI window.

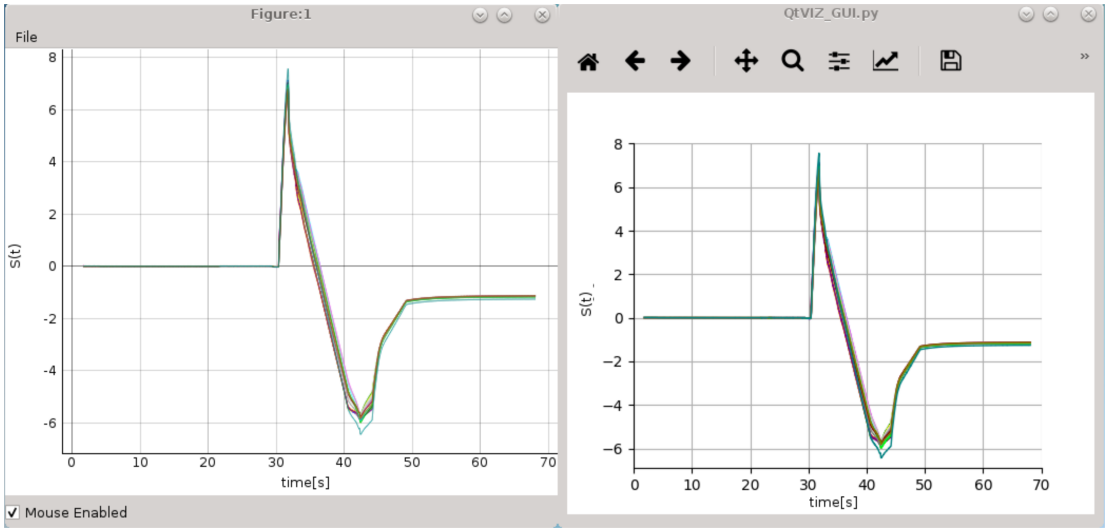


Fig. 28: Comparison of IMASViz *Plot Figure* and *matplotlib window*

2.4.2 Adding a plot to existing figure

The procedure of adding a plot to an already existing figure is as follows:

1. From the previous navigation tree, navigate to the wanted node, for example **ids.magnetics.flux_loop[16].flux.data**
2. Right-click on the node.
3. From the pop-up menu, navigate and select *Plot <node name> to* -> *Figure* -> *Figure:0*

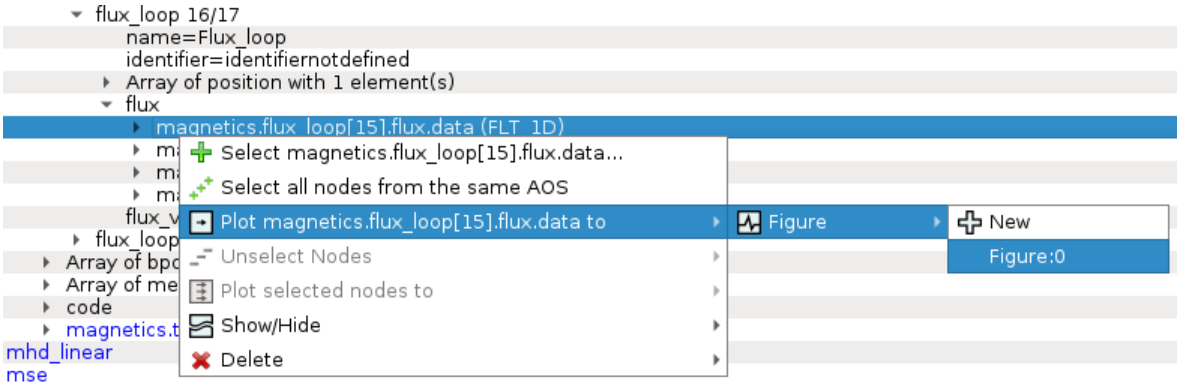


Fig. 29: Plotting to existing figure.

The plot will be added to the selected existing plot as shown in the image below.

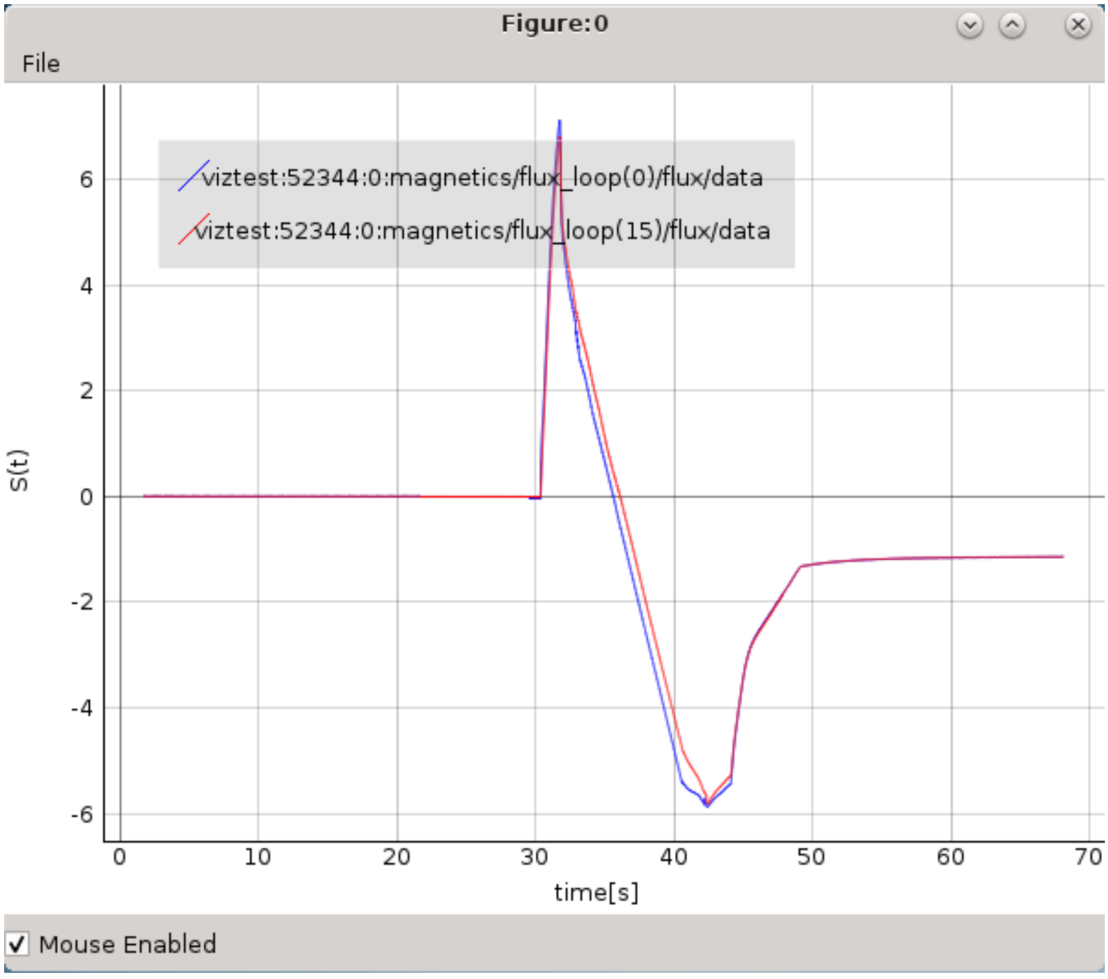


Fig. 30: Plotting to existing figure - result.

2.4.3 Comparing plots between two IDS databases

IMASViz allows comparing of FLT_1D arrays between two different IDS databases (different shots too). The procedure is very similar to the one presented in the section *Adding a plot to existing figure*:

1. Open another IMAS database, same as shown in section *Loading IDS from IMAS local data source*. In this manual this will be demonstrated using IDS with *shot* **52682** and *run* **0** parameters.

Manual IDS case	
parameters	values
User name	g2penkod
IMAS database name	viztest
Shot number	52682
Run number	0

2. Load **occurrence 0** of **magnetics** IDS
3. Navigate through the IDS search for the wanted node, for example **ids.magnetics.flux_loop[0].flux.data**.
4. Right-click on the node.
5. From the pop-up menu, navigate and select *Plot <node name> to* -> *Figure* -> *Figure:0*

The plot will be added to the existing plot as shown in the image below.

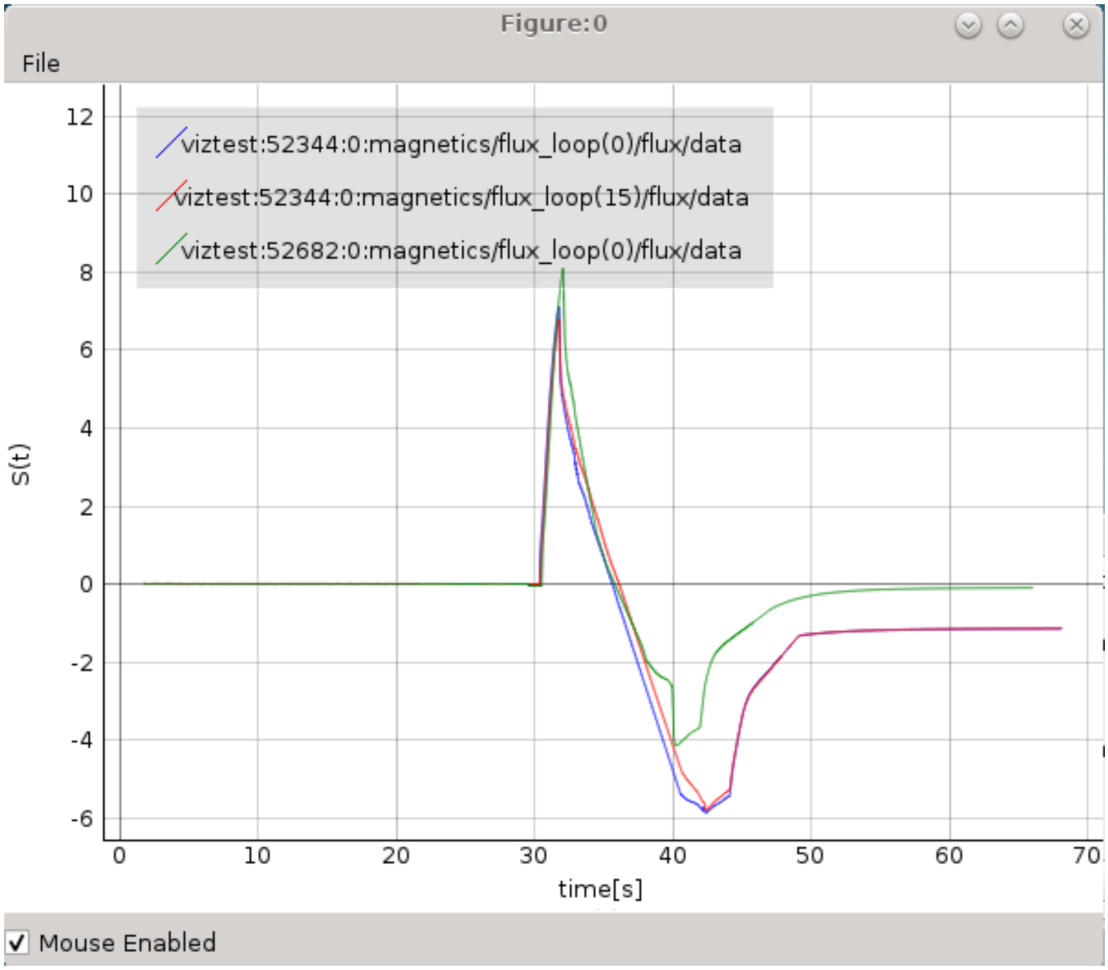


Fig. 31: Plotting from other IDS to existing figure - result.

2.4.4 Plotting a selection of 1D arrays to figure

The procedure of 1D arrays selection and plotting to figure is as follows:

1. In main tree view window set a selection of nodes holding 1D arrays.

> **Note:** How to create a selection of arrays is described in section *Node selection features*.

2. When finished with node selection, either: - right-click on any FLT_1D node, or - click *Node Selection* menu on menubar of the main tree view window.
3. From the pop-up menu, navigate and select *Plot selected nodes to* -> *Figure* -> *New* -> *This IMAS database*.

> **Note:** The same procedure applies plotting the selection to an existing figure.

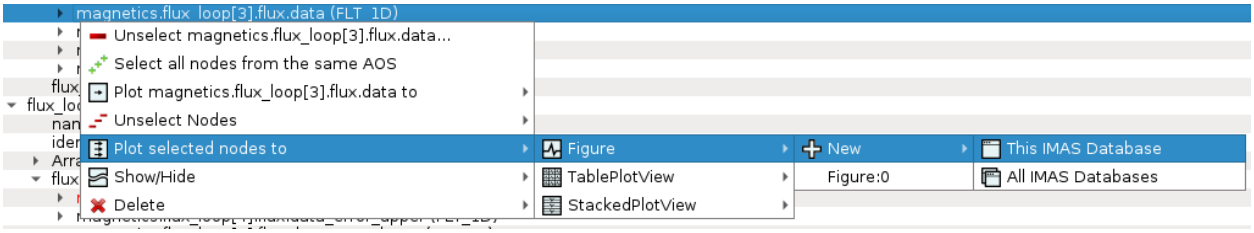


Fig. 32: Plotting selection to a new figure using selection from the currently opened IDS database.

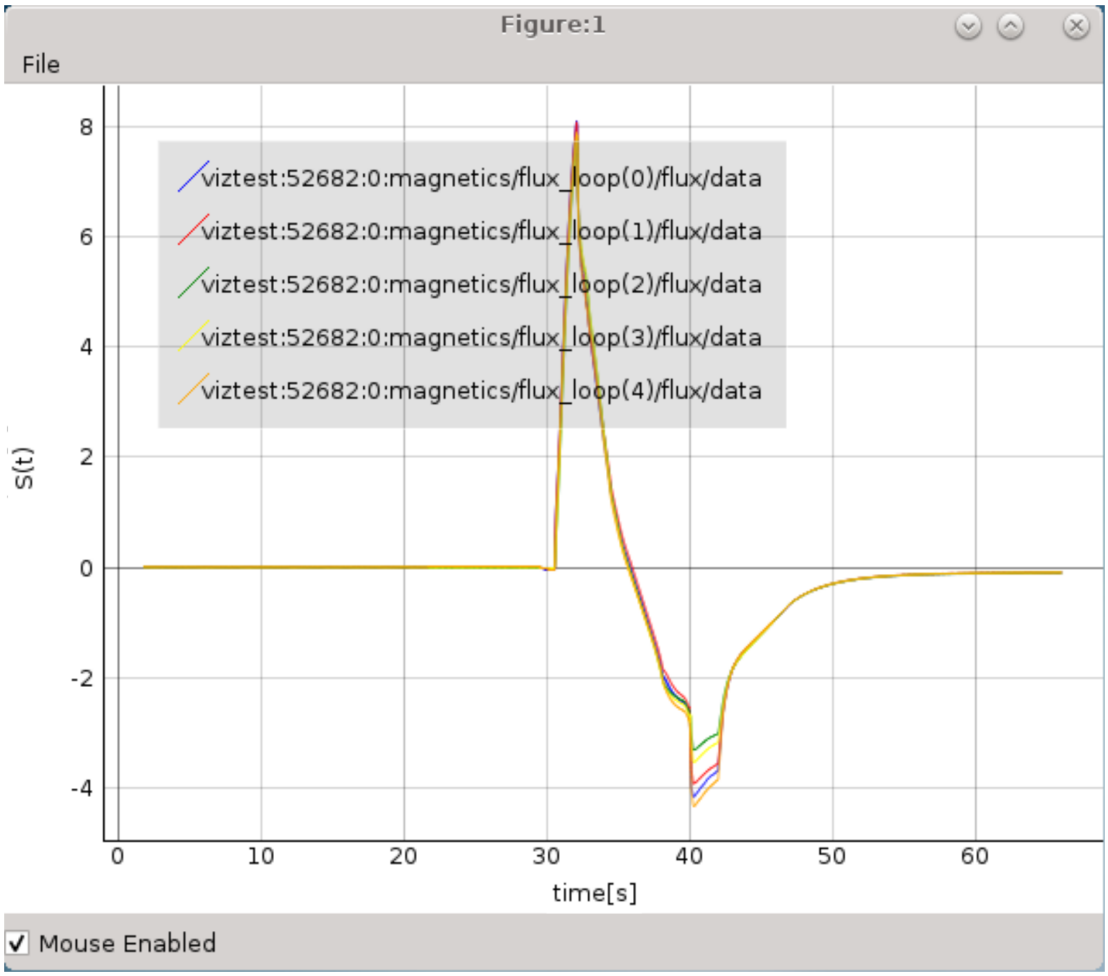


Fig. 33: Example of plot figure, created by plotting data from node selection.

2.4.5 Plotting 1D array as a function of coordinate1 along the time axis

One of the IMASViz features is plotting coordinate along the time axis. This is allowed for IDS nodes, located within **time_slice[:]** structure, and it is already set as a default plotting feature for such arrays.

The procedure to plot such 1D array is quite identical as in section *Plotting a single 1D array to plot figure*. The procedure is described and demonstrated on **equilibrium.time_slice[0].profiles_1d.phi** (Torodial Flux) array.

1. Navigate through the **equilibrium IDS** and search for the time slice node containing **FLT_1D** data, for example **equilibrium.time_slice[0].profiles_1d.phi**.

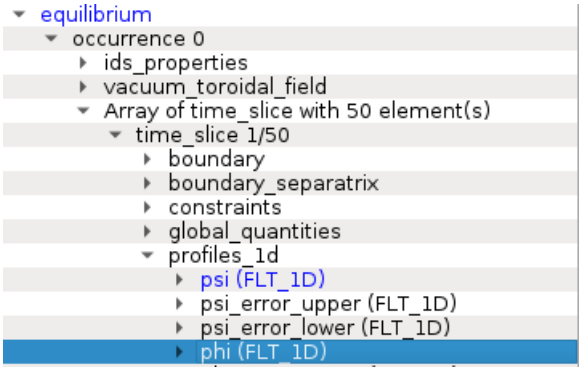


Fig. 34: Example of plottable FLT_1D time slice node.

2. Right-click on the **equilibrium.time_slice[0].profiles_1d.phi (FLT_1D)** node.
3. From the pop-up menu, select the command *Plot equilibrium.time_slice[0].profiles_1d.phi to* -> *figure* -> *New*.

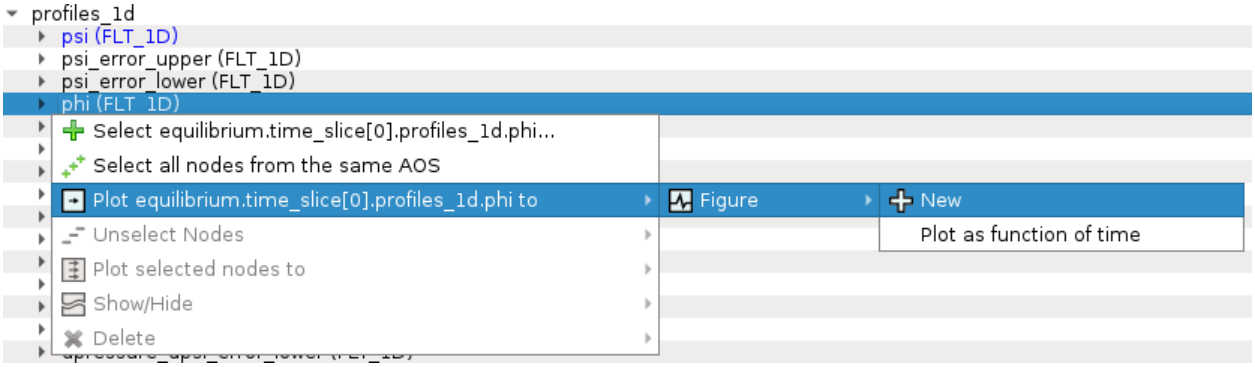


Fig. 35: Navigating through right-click menu to plot data to plot figure.

The plot should display in plot figure as shown in the image below. Note that **coordinate1 = ids.equilibrium.time_slice[0].profiles_1d.psi** for this FLT_1D data array.

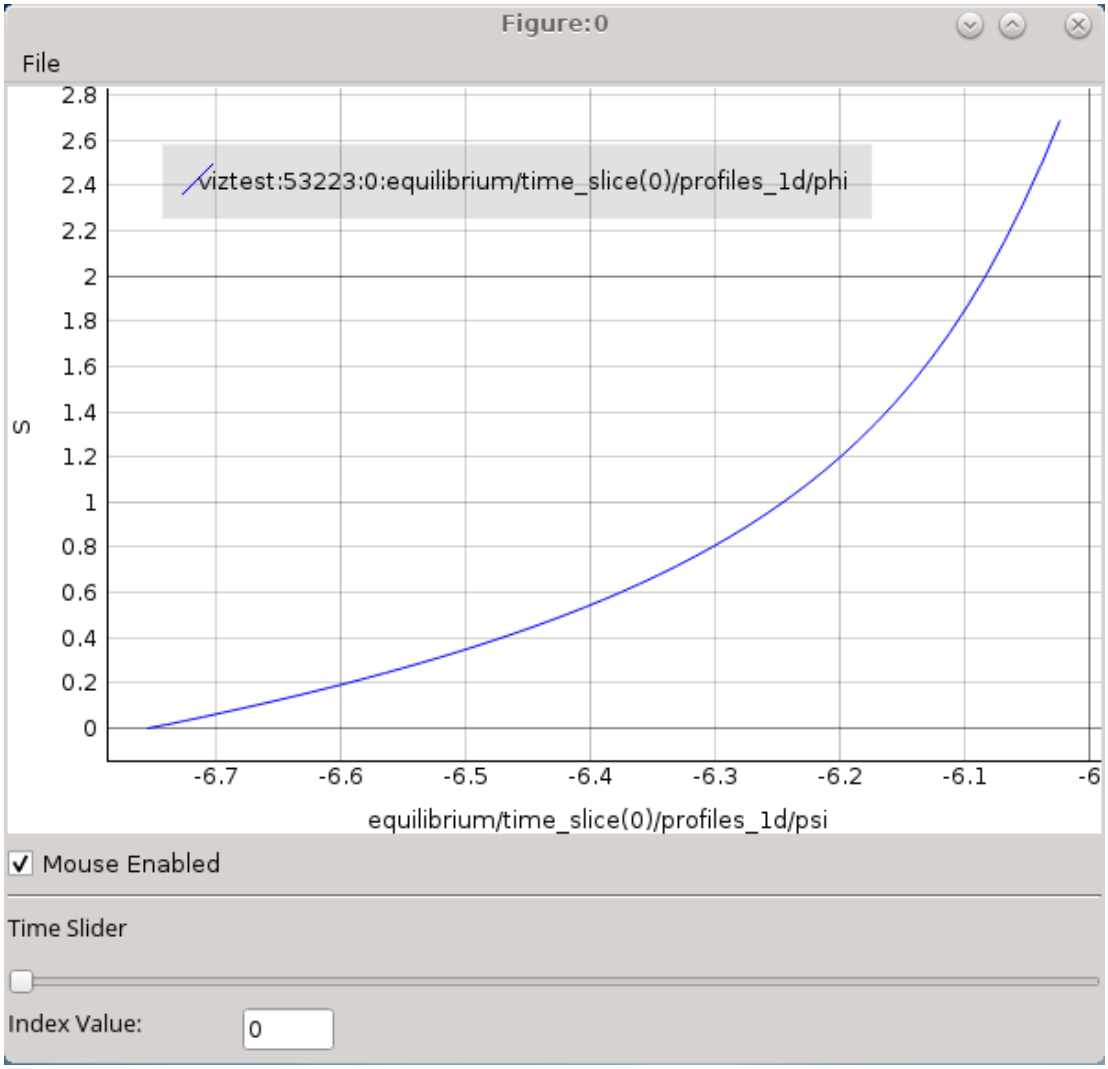


Fig. 36: Time slice plot figure display. The data are represented as a function of coordinate1 (**ids.equilibrium.time_slice[0].profiles_1d.psi**) for the first **phi** time slice (**ids.equilibrium.time_slice[0].profiles_1d.phi**).

The time slider allows you to move along the time axis and the plot will change accordingly.

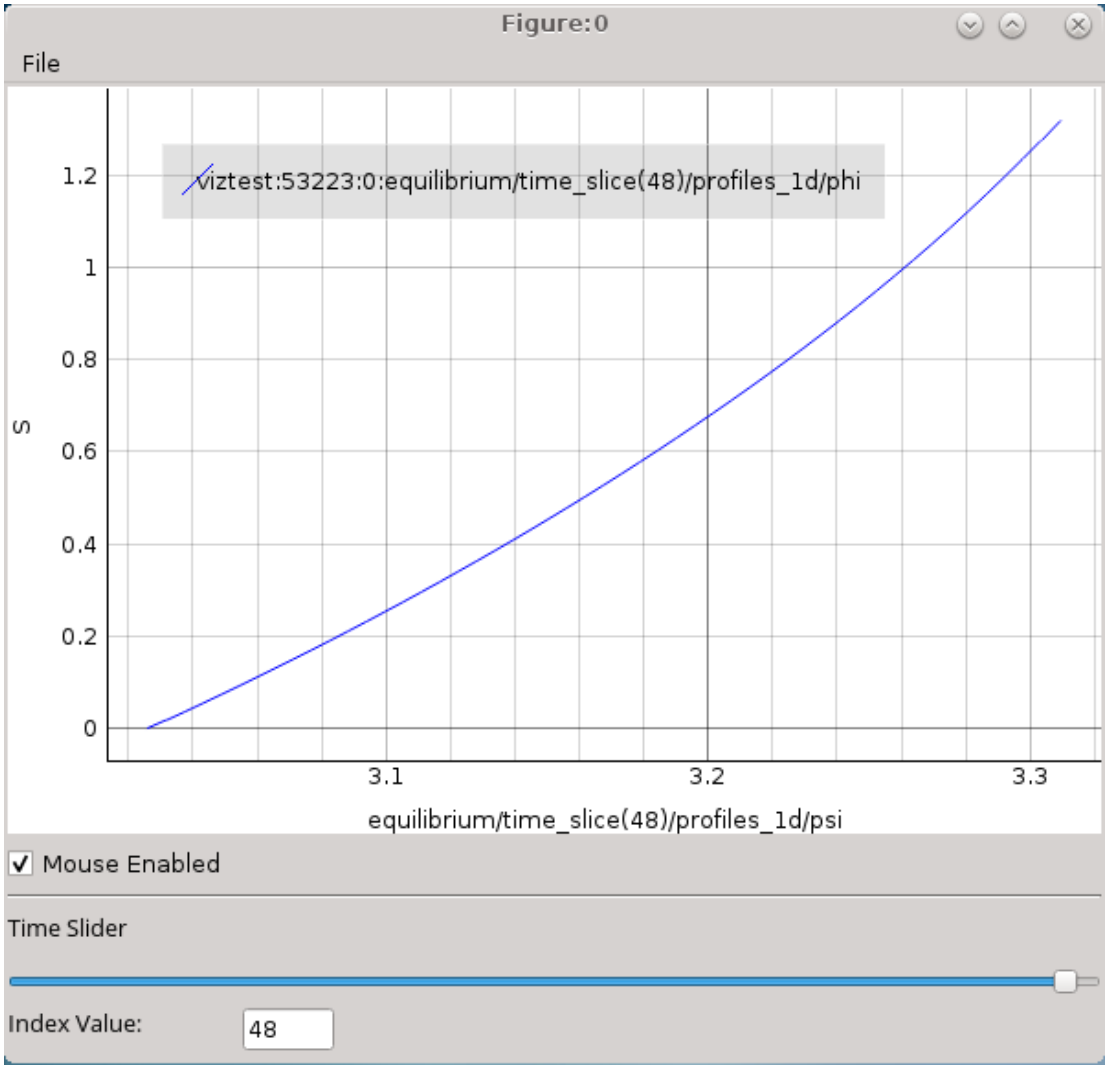


Fig. 37: Time slice plot figure display for **ids.equilibrium.time_slice[48].profiles_1d.phi**. The data are represented as a function of coordinate1 (**ids.equilibrium.time_slice[48].profiles_1d.psi**).

Note: Adding a plot (presented in *Adding a plot to existing figure*) to such existing plot might not work as expected, as the sliding through indexes works directly only the last added plot.

2.4.6 Plotting 1D arrays at index as a function of the time

One of the IMASViz features is plotting array values at certain index as a function of time. This is allowed for IDS nodes, located within **time_slice[:]** structure, and it is already set as a default plotting feature for such arrays.

The procedure is described and demonstrated on **equilibrium.time_slice[0].profiles_1d.f** (Torodial Flux) array.

1. Navigate through the **equilibrium IDS** and search for the time slice node containing **FLT_1D** data, for example **equilibrium.time_slice[0].profiles_1d.f**.

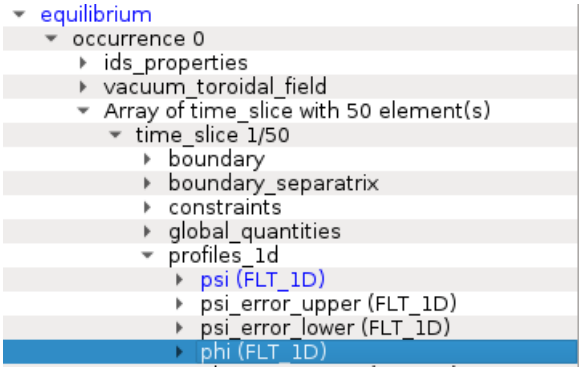


Fig. 38: Example of plottable FLT_1D time slice node.

2. Right-click on the **equilibrium.time_slice[0].profiles_1d.f (FLT_1D)** node.
3. From the pop-up menu, select the command *Plot equilibrium.time_slice[0].profiles_1d.f to* -> *figure* -> *New*.

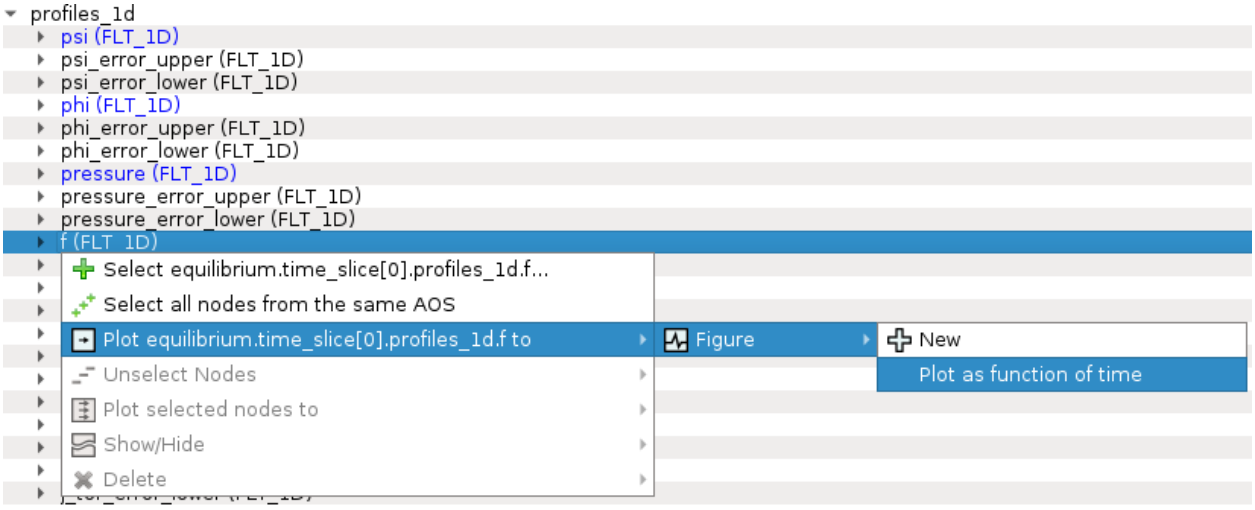


Fig. 39: Navigating through right-click menu to plot data to plot figure.

The plot should display in plot figure as shown in the image below. Note that Y-axis values are an array of **equilibrium.time_slice[:].profiles_1d.f[0]** values through all time slices (marked by **[:]**) and X-axis values are time values found in **equilibrium.time**.

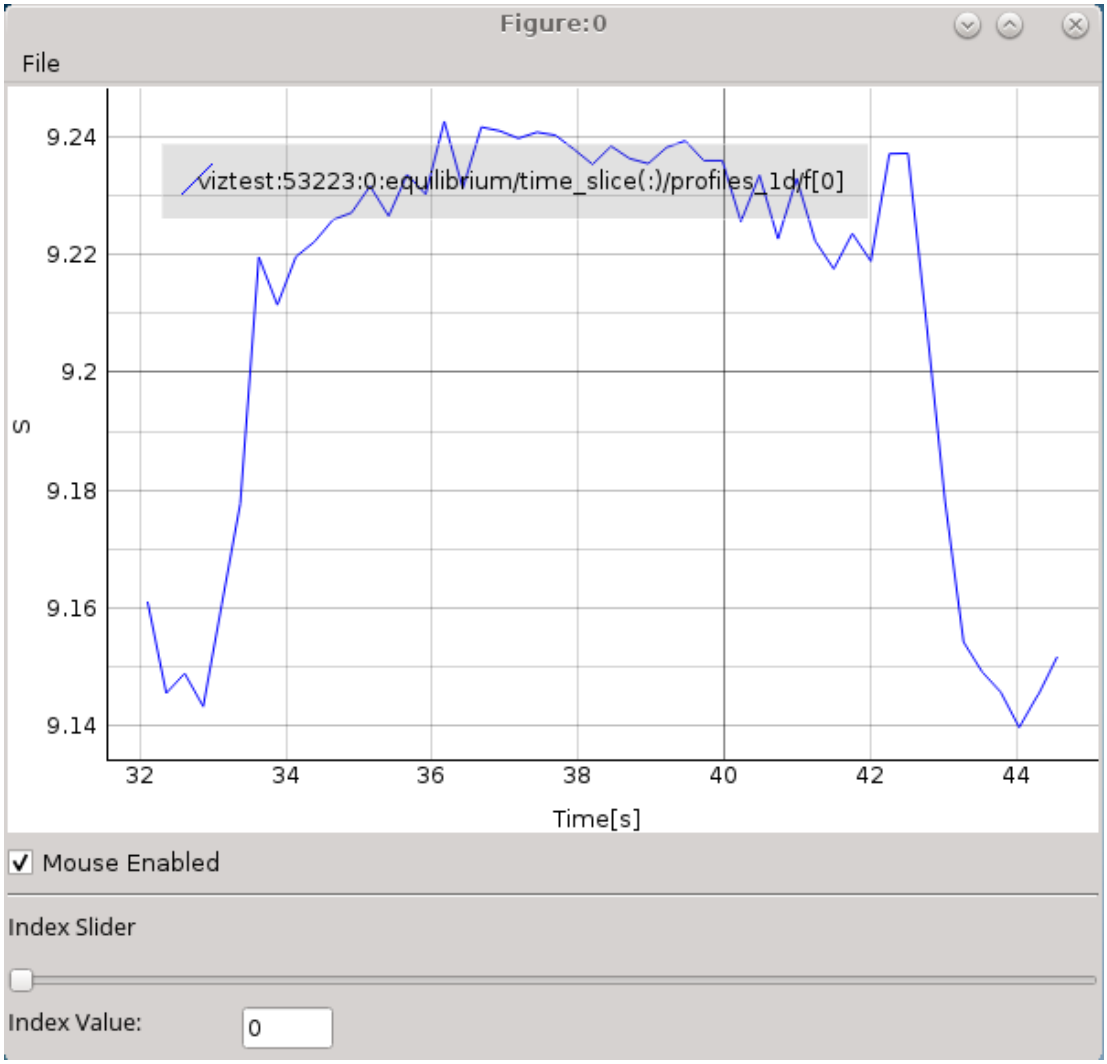


Fig. 40: Plot as a function of time. Y-axis values are an array of **equilibrium.time_slice[:].profiles_1d.f[0]** values through all time slices (marked by **[:]**) and X-axis values are time values found in **equilibrium.time**.

The time slider allows you to move along the array index and the plot will change accordingly.

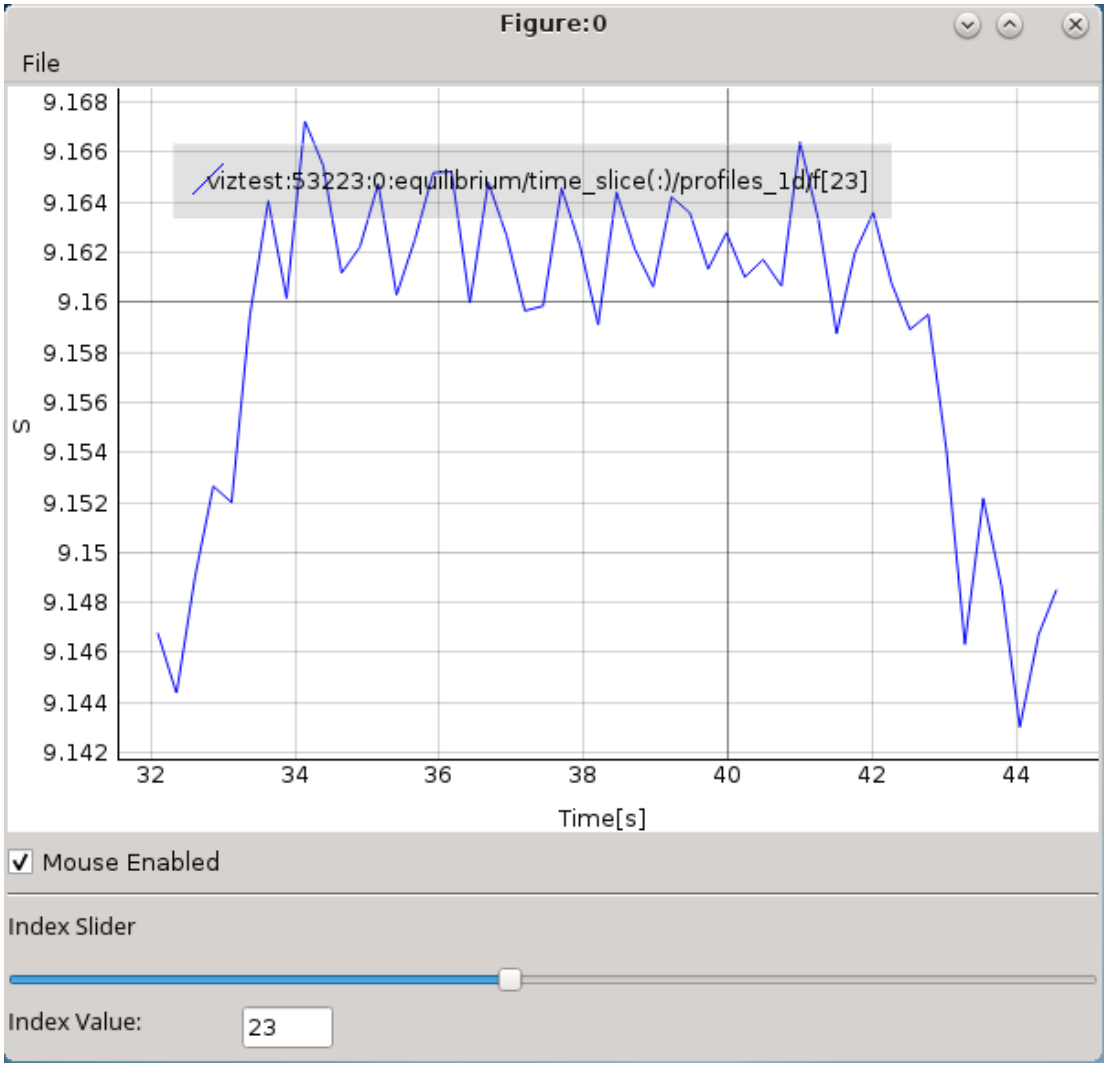


Fig. 41: Plot as a function of time. Y-axis values are an array of **equilibrium.time_slice[:].profiles_1d.f[23]** values through all time slices (marked by **[:]**) and X-axis values are time values found in **equilibrium.time** node (array of FLT_1D values).

Note: Adding a plot (presented in *Adding a plot to existing figure*) to such existing plot might not work as expected, as the sliding through indexes works directly only the last added plot.

2.4.7 MultiPlot features

IMASViz provides few features that allow plotting each signal from a selection to its own plot view window.

Currently there are two such features available:

- *Table Plot View* and
- *Stacked Plot View*.

Each of those Plot Views feature its own plot display layout and plot display window interaction features.

Note: In the old IMASViz, the *Table Plot View* is known as *MultiPlot* and the *Stacked Plot View* is known as *SubPlot*. The decision to rename those features was made due to the previous names not properly describing the feature itself and both of those features being a form of 'MultiPlot' - a window consisting of multiple plot displays.

2.4.7.1 Table Plot View

Table Plot View allows the user to create a multiplot window by plotting every array from selection to its own plot display. The plot displays are arranged to resemble a table layout, as shown in figure below.

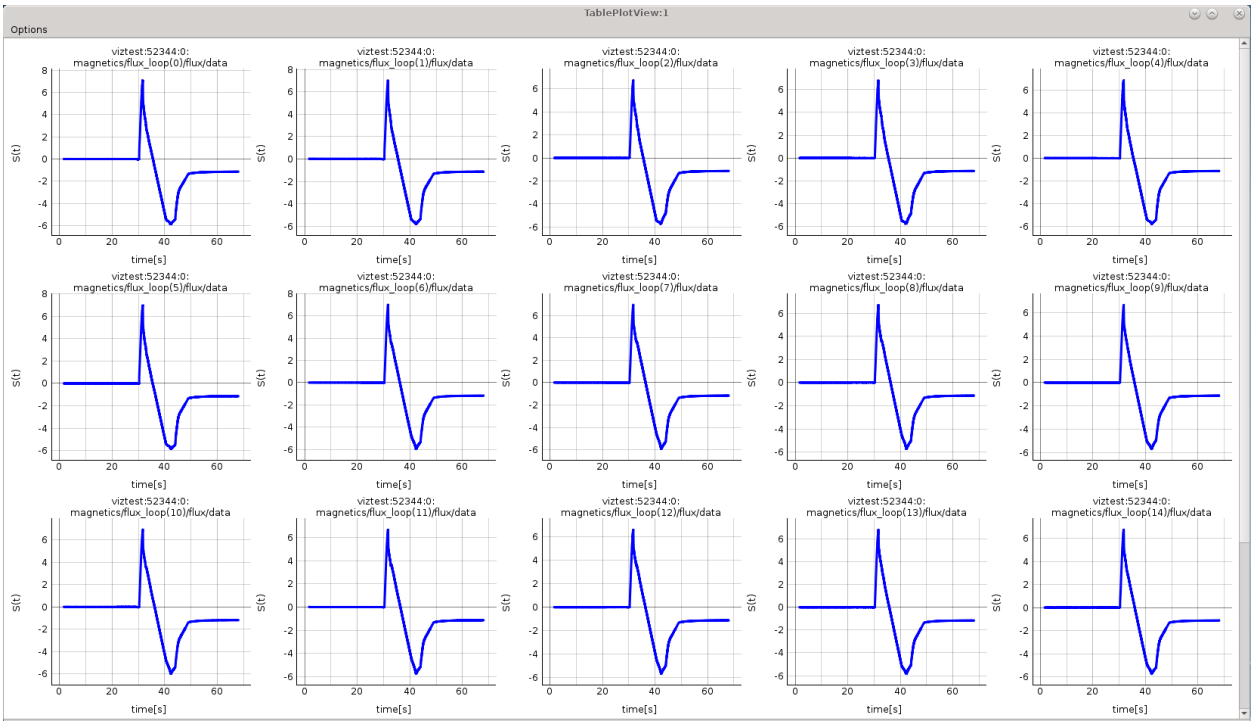


Fig. 42: MultiPlot - *Table Plot View* Example.

Creating New View

To create a new *Table Plot View*, follow the next steps:

1. Create a selection of nodes, as described in section *Plotting a selection of 1D arrays to figure*.
2. **When finished with node selection, either:**
   - right-click on any **FLT_1D** node or
   - click *Node Selection* menu on menubar of the main tree view window.
3. From the pop-up menu, navigate and select *Plot selected nodes to* -> *TablePlotView* -> *New* -> *This IMAS database* or *All IMAS databases* or.

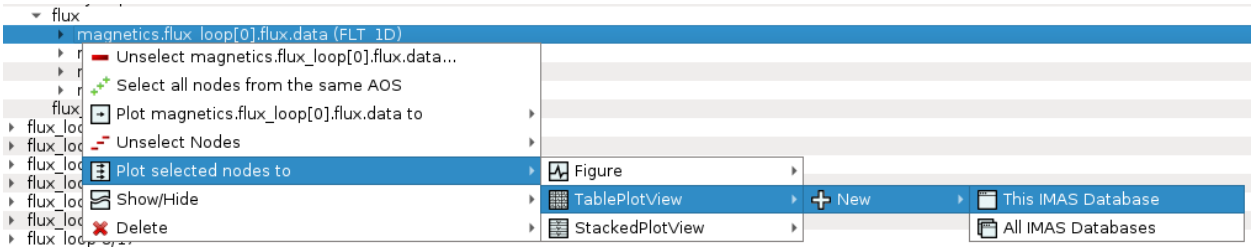


Fig. 43: Plotting selection to a new figure using selection from the currently opened IDS database.

The *Table Plot View* window will then be created and shown.

Note: Each plot can be customized individually by right-clicking to the plot d display and selecting option *Configure Plot*.

Note: Scrolling down the *Table Plot View* window using the middle mouse button is disabled as the same button is used to interact with the plot display (zoom in and out). Scrolling can be done by clicking the scroll bar on the right and dragging it up and down.

Save MultiPlot Configuration

MultiPlot configuration (currently available only for *Table Plot View* feature) allows the user to save the MultiPlot session and load it later.

To create MultiPlot configuration, follow the next steps:

1. Create a selection of nodes, as described in section *Plotting a selection of 1D arrays to figure*.
2. Create a *Table Plot View*, as described in *Table Plot View*.
3. In *Table Plot View* menubar navigate to **Options -> Save Plot Configuration**

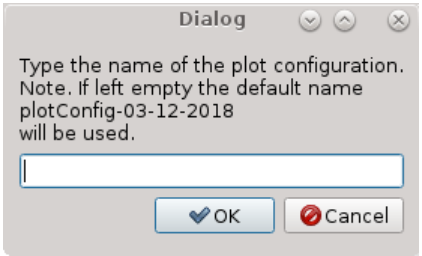


Fig. 44: Save Plot Configuration Dialog Window.

4. Type configuration name in the text area.
5. Press **OK**.

Note: The configurations files are saved to `$HOME/.imasviz` folder.

Applying MultiPlot configuration to other IMAS database

To apply MultiPlot configuration to any IMAS database, follow the next steps:

1. Open IMAS database.
2. In Main Tree View Window menu navigate to **Actions** -> **Apply Configuration**. In the shown window switch to *Apply Plot Configuration* tab.

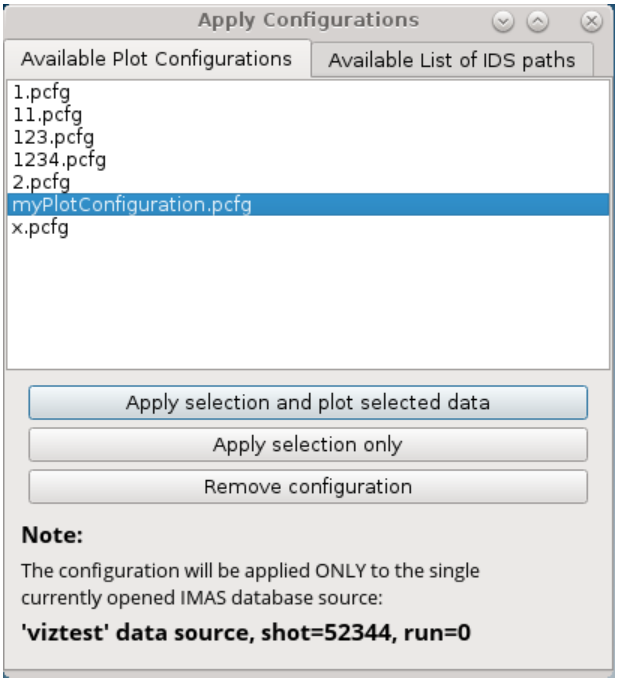


Fig. 45: Apply Plot Configuration GUI Window.

3. Select the configuration from the list.
4. Press **Apply selection and plot selected data**.

   The *Table Plot View* will be created using the data stored in the configuration file.

Note: Currently this feature will take all plot data from single (currently) opened IMAS database, event though MultiPlot configuration was made using plots from multiple IMAS databases at once. This feature is to be improved in the future.

Warning: The plots order depends on the order in which the data selection has been performed. First selected data will be the first plots in the *Table Plot View* window.

2.4.7.2 Stacked Plot View

Stacked Plot View allows the user to create a multiplot window by plotting every array from selection to its own plot display. The plot displays are arranged to resemble a stack layout, as shown in figure below. All plots displays always

share the same X and Y range, even when using plot interaction features such as *Zoom in/out*, *Pan Mode*, *Area Zoom Mode* etc.

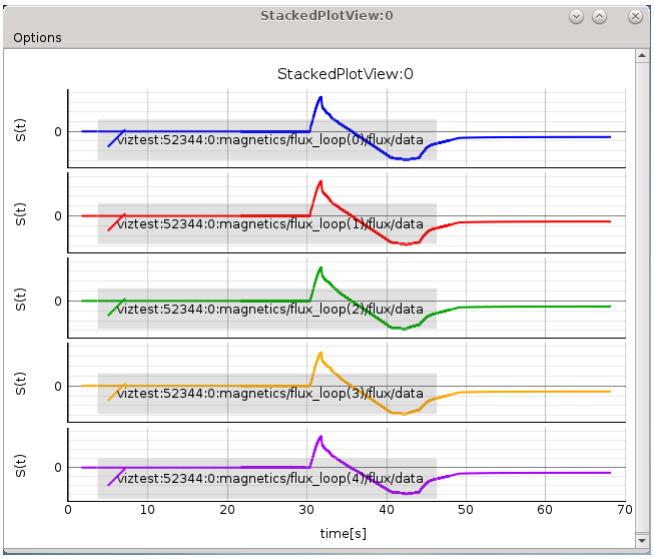


Fig. 46: MultiPlot - *Stacked Plot View* Example.

Creating New View

To create a new *Stacked Plot View*, follow the next steps:

1. Create a selection of nodes, as described in section *Plotting a selection of 1D arrays to figure*.
2. **When finished with node selection, either:**
   - right-click on any FLT_1D node or
   - click *Node Selection* menu on menubar of the main tree view window.
3. From the pop-up menu, navigate and select *Plot selected nodes to* -> *StackedPlotView* -> *New* -> *This IMAS database* or *All IMAS databases* or.

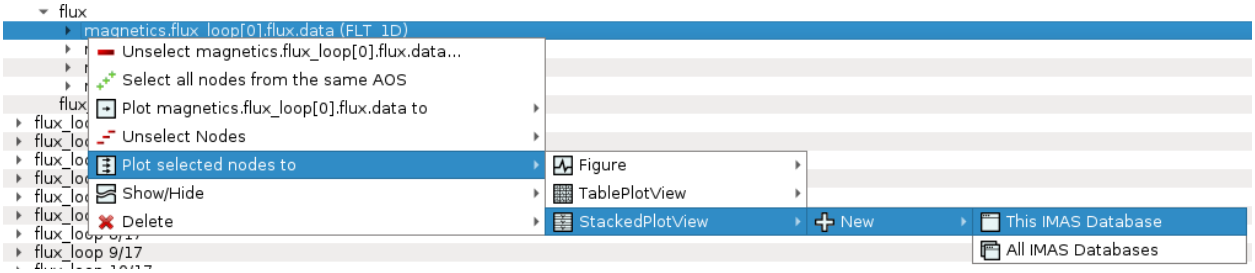


Fig. 47: Plotting selection to a new figure using selection from the currently opened IDS database.

The *Stacked Plot View* window will then be shown.

Note: Each plot can be customized individually; right click on a node and select 'Configure Plot'.

2.5 Plugins

IMASViz allows the use of custom-made plugins which further extend the IMASViz functionality. The plugins currently available are listed here together with a manual section on how to use them.

2.5.1 Equilibrium overview plugin

This subsection describes and demonstrates the use of IMASViz **equilibrium overview plugin**, a simple utility for loading and displaying multiple data from **equilibrium IDS** at once.

2.5.1.1 Executing the equilibrium overview plugin

The procedure of executing the IMASViz **Equilibrium overview plugin** is as follows:

1. In opened IMAS database navigate to the **equilibrium** IDS.
2. While holding the **shift keyboard button**, **right-click** on the **equilibirum** node. In a pop up menu a list of available plugins for the equilibrium IDS will be displayed.
3. Click the *Equilibrium overview* option.

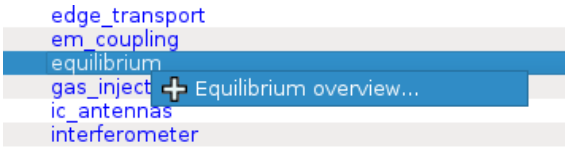


Fig. 48: Equilibrium overview plugin popup menu option.

The plugin will then first grab the data from the Equilibrium IDS which takes a few seconds. After that the plugin window will appear.

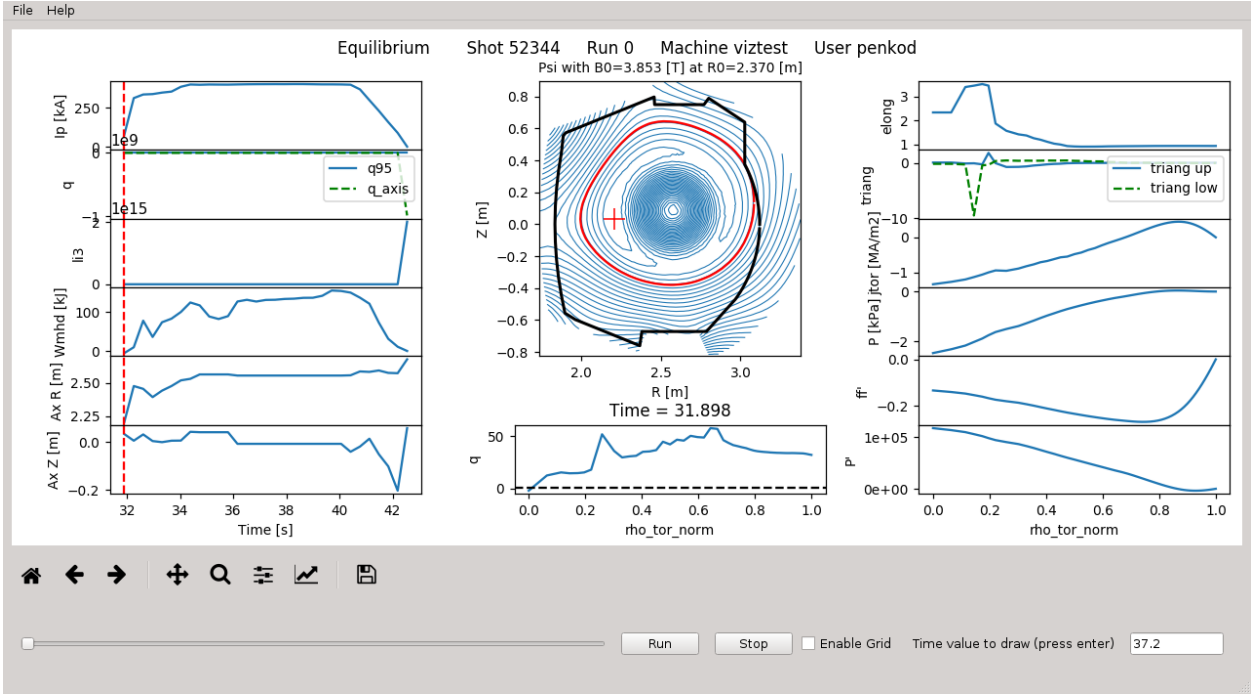


Fig. 49: Equilibrium overview plugin main window.

2.5.1.2 Equilibrium overview plugin GUI features

The Equilibrium plugin allows some interaction with the plots that are described below.

Plot at time slice using slider

The plotting time value can be easily set by **holding** and **moving** the slider. By doing that the **red dashed line** on the left plot will move at the same time, indicating the selected time value (position).

On **slider release** all plots are updated for the selected time.

Plot at time slide using time value

The time can be specified directly by typing the value to the **time value box**.

Fig. 50: Time value box.

By pressing *enter* keyboard button, all plots are be updated for the specified time.

Note: Check the left plot for the time values range.

Run through time slices

This feature allows the plugin to go through all time values and at the same time update all plots, giving a good overall overview.

This is done by pressing the *Run* button.

Enable plot grid

To enable grid on all plots check the *Enable Grid* option.

2.5.2 SOLPS overview plugin

This subsection describes and demonstrates the use of IMASViz **SOLPS overview plugin**, a simple utility for loading and displaying grid geometry (including grid subsets), and physics quantities (such as electron density, ion temperature etc.) stored within **edge_profiles IDS GGD (General Grid Description)**.

Note: The **SOLPS overview plugin** can be run also in a standalone mode (outside IMASViz) either from widget source code or from plugin .ui source. The necessary sources can be found in directory `$VIZ_HOME/imasviz/VizPlugins/viz_solps`. There, to run standalone widget run command `python3 SOLPSwidget.py` in terminal. To run standalone plugin (the same one as used in IMASViz) run command `python3 run_plugin_ui_standalone.py`.

Note: The development procedure of the **SOLPS overview plugin** can be seen in a short movie found in section *Developing a custom user interface (UI) plugins with Qt designer*.

2.5.2.1 Executing the SOLPS overview plugin

The procedure of executing the IMASViz **SOLPS overview plugin** is as follows:

1. In opened IMAS database navigate to the **edge_profiles** IDS.
2. While holding the **shift keyboard button**, **right-click** on the **edge_profiles** node. In a pop up menu a list of available plugins for the edge_profiles IDS will be displayed.
3. Click the *SOLPS overview* option.

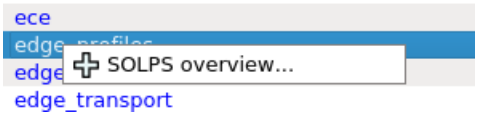


Fig. 51: SOLPS overview plugin popup menu option

After that the main plugin window will appear, containing an empty plot widget and a few buttons.

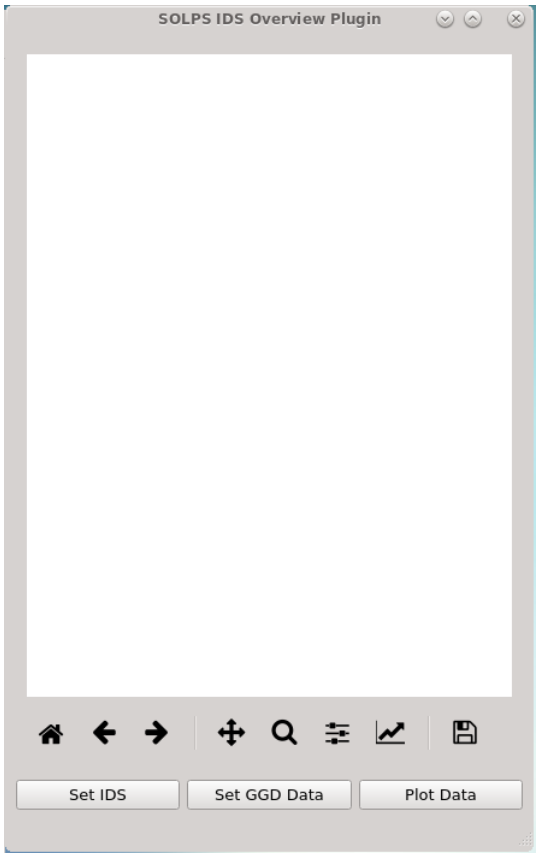


Fig. 52: SOLPS overview plugin main window.

4. Click the *Set IDS* button. The plugin will then first read the available data from the Edge Profiles IDS (provided by IMASViz) and build the tree view which takes a few seconds.
5. Click the *Set Data* button. After that a dialog window will appear, requesting:
    - *GGD Grid (Slice)*, specifying a grid geometry time slice. In most cases the grid geometry does not change with time so in such cases is obsolete to 're-write' it (that is also the reason why the **GGD Grid (grid_ggd)** and **GGD Quantities (ggd)** structures are separated).
    - *GGD Quantities (Slice)*, specifying the time slice for physics quantities,
    - *Grid Subset*, listing all available 2D grid subsets for specified **GGD Grid** and **GGD Quantities** slice.
    - *Grid Subset Quantity*, listing all available quantities for grid subset specified by *Grid Subset* drop down list.

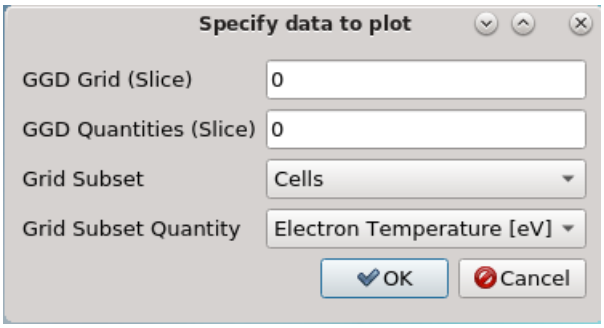


Fig. 53: SOLPS overview plugin dialog for setting data (basic example values are set).

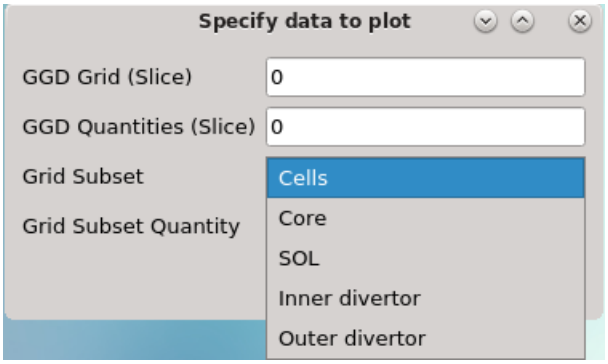


Fig. 54: List of available (2D) grid subsets for current IDS.

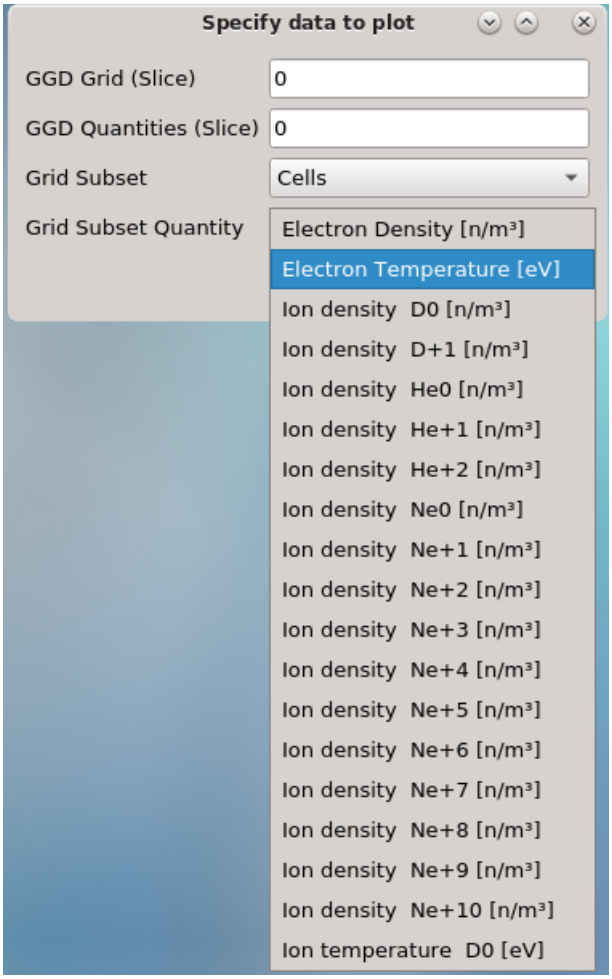


Fig. 55: List of available physics quantities for current IDS.

After the requested parameters are set, press the *OK* button.

6. Click the *Plot Data* button. After pressing the button the plot widget will be populated with plot created using the specified data.

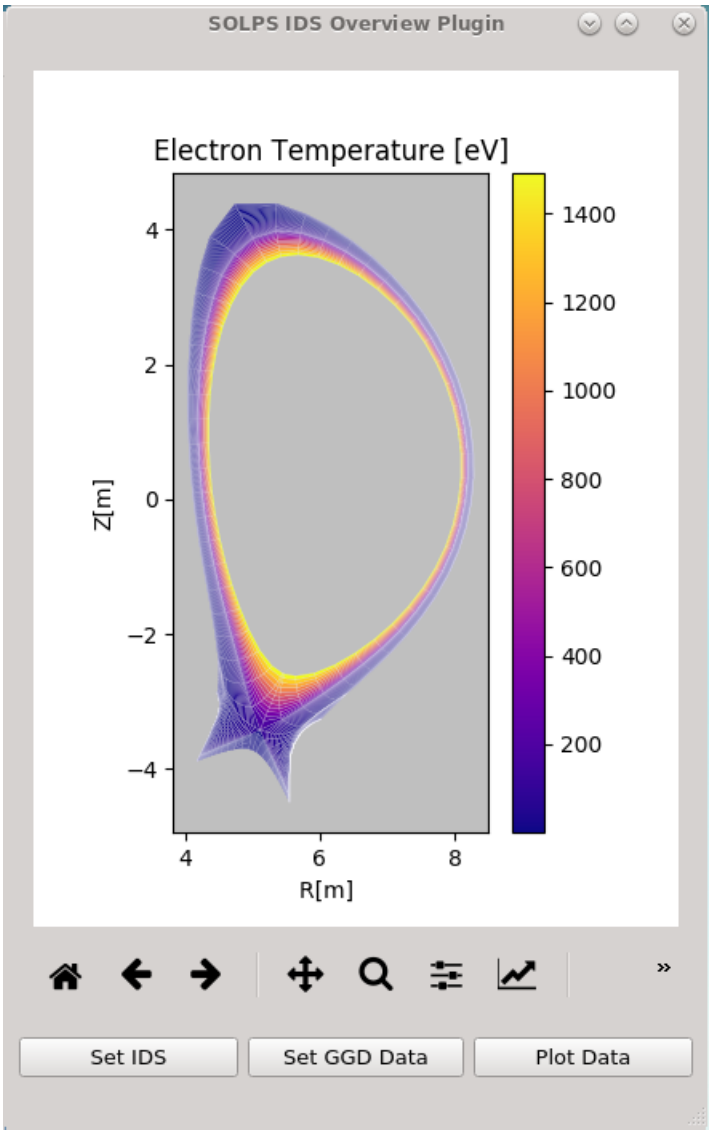


Fig. 56: SOLPS overview plugin plot - **Cells** grid subset (all 2D quad elements in the domain) + electron temperature quantity values.

2.6 Other GUI features

Other *IMASViz* GUI features are listed here.

2.6.1 Hide/Show Plot Window

To **hide** or **show** any of the plot windows (*Figure*, *Table Plot View* etc.), first right-click on any signal node. From the shown pop-up menu, select the command *Show/Hide* and select the plot window you wish to show or hide.

Note: Shown plot windows will be hidden while hidden windows will be shown.

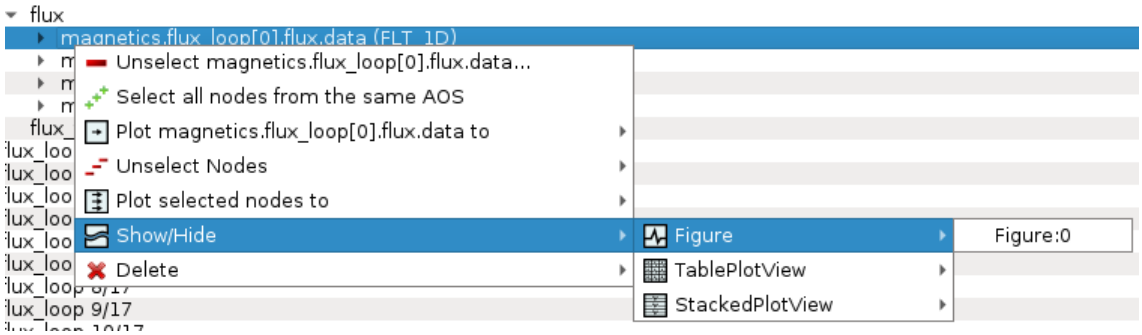


Fig. 57: Show/Hide plot window.

2.6.2 Delete Plot Window

To delete any of the plot windows, first right-click on any signal node. From the shown pop-up menu, select the command *Delete* and select the plot window you wish to delete or delete all plot windows of certain type (*Figure*, *Table Plot View* etc.).

Warning: The plot window will be deleted permanently.

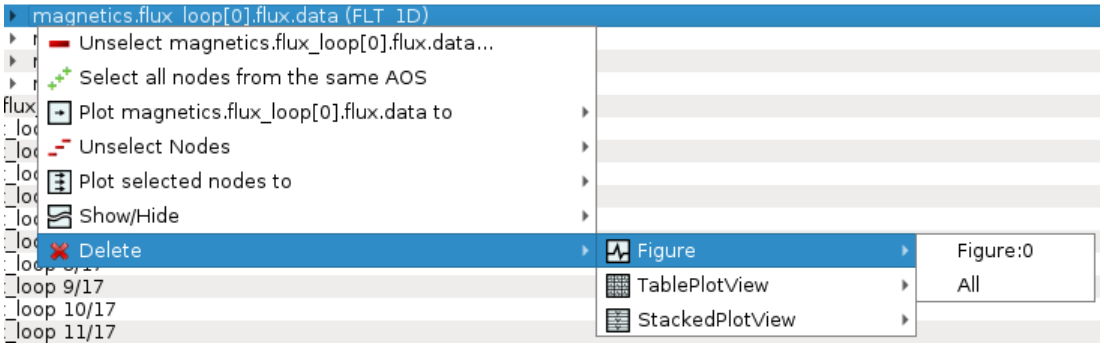


Fig. 58: Delete plot window.

2.6.3 Export IDS

This feature allows the user to export the opened IDSs to a separate IDS with custom parameters.

Note: Only IDSs that are currently opened and populated in the tree view window are exported using this feature.

Note: IMAS database (e.g. device or machine) must exist before trying to export IDS to it.

To export the IDSs, first navigate to the menu *Actions -> Export to IDS*.

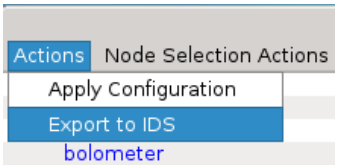


Fig. 59: *Export to IDS* feature in the menu bar.

A popup dialog will be shown where the IDS case parameters are set.

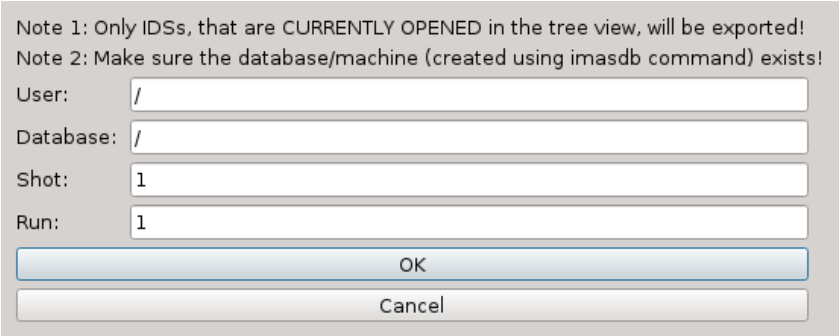


Fig. 60: Empty dialog of the *Export to IDS* feature.

The IDS case parameters are described below:

GUI Fields	Description
User name	Creator/owner of the IMAS IDSs database
IMAS database name	IMAS database label, usually device/machine name of the IMAS IDS database (i. e. iter, aug, west. . .)
Shot number	Pulse shot number
Run number	Pulse run number

When the case parameters are set, press the *OK* button to export to the specified new IDS.

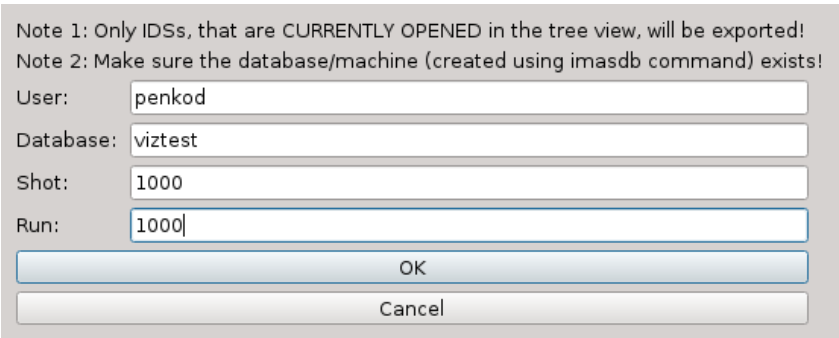


Fig. 61: *Export to IDS* feature dialog with set parameters.

The export status gets displayed in the tree view log widget.

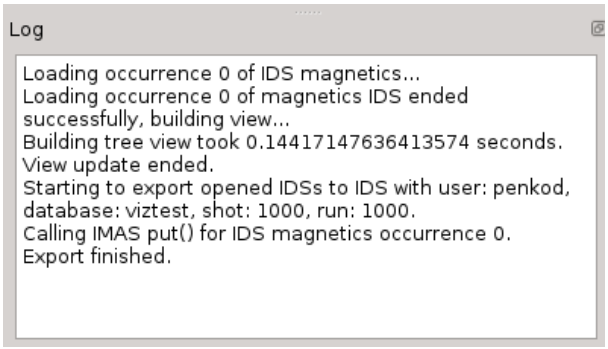


Fig. 62: *Export to IDS* finish message.

2.6.4 Export plot

This feature allows the user to export:

- plot to image,
- plot to .SVG format,
- plot to matplotlib window, and
- plot data to CSV or HDF5 format.

This feature is accessible by right-clicking anywhere inside the plot window and by selecting *Export...* option in the shown popup menu.

Fig. 63: Selections of the *Export* feature in the popup menu.

A dialog window will be shown, displaying the possible export options. Some the export options allow also setting basic parameters such as image size etc.

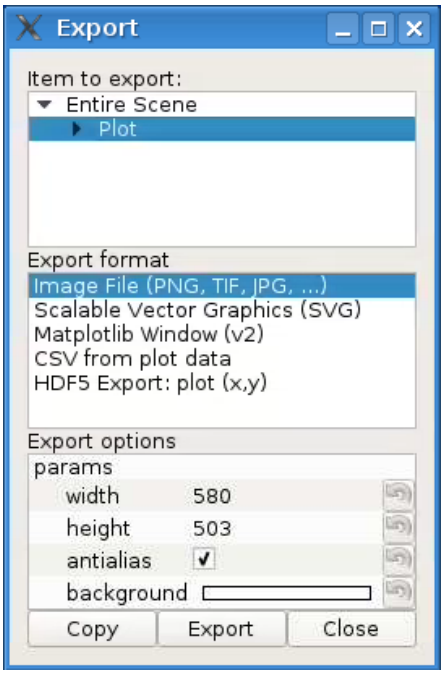


Fig. 64: *Export* feature dialog window.

2.6.5 Hide/Show/Delete DTV

This feature allows the user to either hide, show or delete existing **Data Tree Views** (DTVs).

This feature is available from the starting dialog window (the one where the database parameters **user/database/shot/run** are being set) by right clicking anywhere inside the window. A popup menu will be shown displaying those available options.

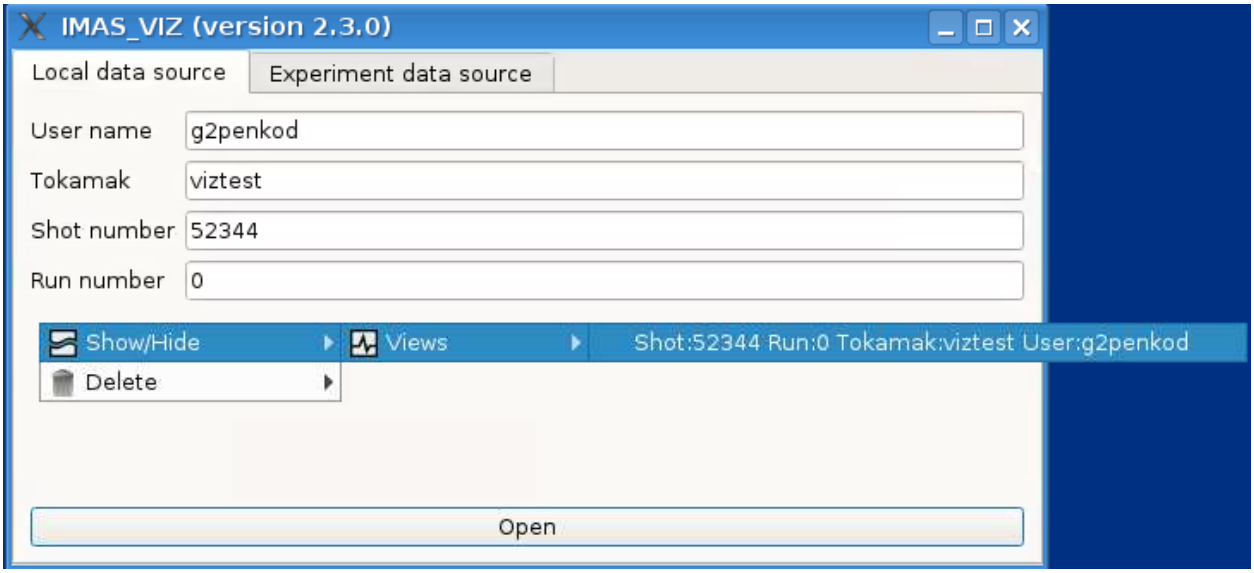


Fig. 65: *Show/Hide/Delete DTV* popup menu.

2.7 Scripting

There are two main methods to open and browse IMAS IDS database using *IMASViz*. First is the standard way of running IMASViz (as described in *Getting Started*) and setting the parameters in the GUI, the second is through a script. Furthermore, some GUI actions (like node selection, plot action commands etc.) can be performed through scripting too. This way a simple IMASViz session can be set and populated on run.

This section describes the second method: how to create and use **user-made** Python3 scripts that run and populate *IMASViz*.

IMASViz scripting can be seen as an advanced alternative to the *Apply Configuration* features (described in *Apply Selection From Saved Configuration* and *Applying MultiPlot configuration to other IMAS database*).

The procedure below can be used with either **IMASViz module** and **IMASViz from source** but the environment **must be set accordingly to the method the IMASViz is used (!)** (as described in *Running IMASViz as a module on The GateWay* and *Running IMASViz from source*).

2.7.1 Adding IMASViz Path to PYTHONPATH

Warning: Before proceeding, make sure that the environment is properly pre-set! This way also the **VIZ_HOME** system variable, required in this manual section, is available.

The IMASViz home directory can be added to **PYTHONPATH** system variable by running in the terminal the command below (use the command that matches your shell - **c-shell** or **bash**):

Note: **PYTHONPATH** is a "list" of paths that tell Python where to look for sources, libraries etc.

```
# c-shell (csh)
setenv PYTHONPATH ${VIZ_HOME}:${PYTHONPATH}
# bash
export PYTHONPATH=${VIZ_HOME}:${PYTHONPATH}
```

2.7.2 Creating A Script

This subsection will cover the basic procedure of writing a Python3 script that can be used with *IMASViz*. Few such working script examples are shown in section *Script examples*. The same examples can be found in project GIT repository here.

1. **First, the imports are required:**
    - constants, functions, and methods of the Python interpreter,
    - PyQt5 classes and routines, and
    - IMASViz classes and routines.

```
# !/usr/bin/python
# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
```

(continues on next page)

(continued from previous page)

```
# IMASViz source imports
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.VizGUI.VizGUICommands.VizMenusManagement.QVizSignalHandling \
    import QVizSignalHandlingds.VizMenusManagement.QVizSignalHandling \
        import QVizSignalHandling
```

2. Set object managing the PyQt GUI application's control flow:

```
app = QApplication(sys.argv)
```

3. Check if necessary system variables are set

```
QVizGlobalOperations.checkEnvSettings()
```

4. Set Application Program Interface

```
api = Viz_API()
```

5. Set data source retriever/factory

```
dataSourceFactory = QVizDataSourceFactory()
```

6. Load IMAS database and build the data tree view

```
f1 = api.CreateDataTree(dataSourceFactory.create(shotNumber=52344,
                                                 runNumber=0,
                                                 userName='g2penkod',
                                                 imasDbName='viztest'))
```

7. Add the build data tree view (DTV) to a list (!)

```
f = [f1]
```

8. Set the list of node paths

```
pathsList1 = []
for i in range(0, 5):
    pathsList1.append('magnetics/flux_loop(' + str(i) + ')/flux/data')
```

9. Select signals corresponding to the list of node paths

```
api.SelectSignals(f1, pathsList1)
```

10. Show the data tree window

```
f1.show()
```

11. Plot selected nodes

```
f = [f1]
api.PlotSelectedSignalsFrom(f)
```

12. Plot data from the first data source (f1) to Table Plot View

```
QVizSignalHandling(f1.dataTreeView).onPlotToTablePlotView(all_DTV=False)
```

13. Plot data from the first data source (f1) to Stacked Plot View

```
QVizSignalHandling(f1.dataTreeView).onPlotToStackedPlotView(all_DTV=False)
```

14. Keep the application running

```
sys.exit(app.exec_())
```

The final script is available below.

```
# !/usr/bin/python
# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.VizGUI.VizGUICommands.VizMenusManagement.QVizSignalHandling \
    import QVizSignalHandling

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
QVizGlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = QVizDataSourceFactory()

# Load IMAS database and build the data tree view
f1 = api.CreateDataTree(dataSourceFactory.create(shotNumber=52344,
                                                 runNumber=0,
                                                 userName='g2penkod',
                                                 imasDbName='viztest'))

# Add data tree view frame to list (!)
f = [f1]

# Set the list of node paths that are to be selected
pathsList1 = []
for i in range(0, 5):
    pathsList1.append('magnetics/flux_loop(' + str(i) + ')/flux/data')

# Select signal nodes corresponding to the paths in pathsList
api.SelectSignals(f1, pathsList1)

# Show the data tree view window
f1.show()
```

(continues on next page)

(continued from previous page)

```
# Plot signal nodes
# Note: Data tree view does not need to be shown in order for this
#       routine to work
api.PlotSelectedSignalsFrom(f)

# Plot data from the data source to Table Plot View
QVizSignalHandling(f1.dataTreeView).onPlotToTablePlotView(all_DTV=False)

# Plot data from the data source to Stacked Plot View
QVizSignalHandling(f1.dataTreeView).onPlotToStackedPlotView(all_DTV=False)

# Keep the application running
sys.exit(app.exec_())
```

2.7.3 Running the script

With the environment set (done in *Adding IMASViz Path to PYTHONPATH*) and script completed (done in *Creating A Script*), the script can be run using the basic Python3 terminal command:

```
python3 <path_to_script>/<script_name>.py
```

By running this script all *Data Tree Views*, *Plot Widgets* and *MultiPlot Views*, previously set in the script, should show, as shown in the figure below.

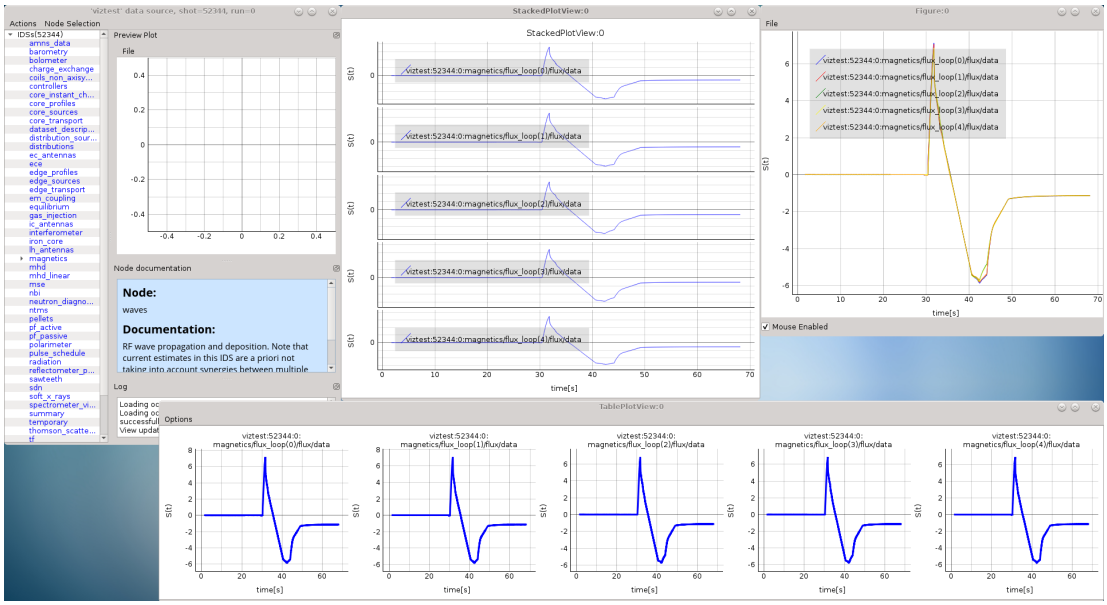


Fig. 66: The result of running the script example: *Data Tree View (DTV)*, *Plot Widget*, *Table Plot View* and *Stacked Plot View* containing multiple plots.

2.7.4 Script examples

Few complete script examples are shown below. The same examples can be found in project GIT repository here.

2.7.4.1 Example 1

```
#!/usr/bin/python
"""This example demonstrates the procedure of plotting multiple arrays to
a single plot, Table Plot View and Stacked Plot View, using IMAS IDS databases
located on the GateWay HPC.
"""

# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizUtils.QVizGlobalValues import QVizGlobalValues

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
QVizGlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = QVizDataSourceFactory()

# Load IMAS database
# dataSource = dataSourceFactory.create(
#                                   dataSourceName=QVizGlobalValues.IMAS_NATIVE,
#                                   shotNumber=54178,
#                                   runNumber=0,
#                                   userName='imas_public',
#                                   imasDbName='west')

# Database on the GateWay HPC (comment the above dataSource code and uncomment
# the one below)
dataSource = dataSourceFactory.create(shotNumber=52344,
                                      runNumber=0,
                                      userName='penkod',
                                      imasDbName='viztest')


# Build the data tree view frame
f = api.CreateDataTree(dataSource)

# Set the list of node paths that are to be selected
paths = []
for i in range(0,6):
    paths.append('magnetics/flux_loop(' + str(i) + ')/flux/data')

# Change it to dictionary with paths an occurrences (!)
```

(continues on next page)

(continued from previous page)

```
paths = {'paths' : paths,
         'occurrences' : [0]}

# Optional: Option with single path in dictionary
# paths = {'paths' : 'magnetics/flux_loop(1)/flux/data'}
# or
# paths = {'paths' : ['magnetics/flux_loop(1)/flux/data']}

# Select signal nodes corresponding to the paths in paths list
api.SelectSignals(f, paths)

# Plot signal nodes
# Note: Data tree view does not need to be shown in order for this routine to
#       work
api.PlotSelectedSignals(f)

# Keep the application running
app.exec()
```

2.7.4.2 Example 2

```
# !/usr/bin/python
"""This example demonstrates the procedure of plotting multiple arrays to a
single plot, Table Plot View and Stacked Plot View, using IMAS IDS databases
located on the GateWay HPC.
"""

# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.VizGUI.VizGUICommands.VizMenusManagement.QVizSignalHandling \
    import QVizSignalHandling

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
QVizGlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = QVizDataSourceFactory()

# Load IMAS database and build the data tree view frame
f1 = api.CreateDataTree(dataSourceFactory.create(shotNumber=52344,
                                                 runNumber=0,
```

(continues on next page)

(continued from previous page)

```
                                                    userName='g2penkod',
                                                    imasDbName='viztest'))

# Load IMAS database and build the data tree view frame
f2 = api.CreateDataTree(dataSourceFactory.create(shotNumber=52682,
                                                    runNumber=0,
                                                    userName='g2penkod',
                                                    imasDbName='viztest'))

# Add data tree view frames to list (!)
f = [f1, f2]
# Set the list of node paths that are to be selected
pathsList1 = []
for i in range(0, 5):
    pathsList1.append('magnetics/flux_loop(' + str(i) + ')/flux/data')
pathsList2 = []
for i in range(0, 6):
    pathsList2.append('magnetics/bpol_probe(' + str(i) + ')/field/data')

# Select signal nodes corresponding to the paths in paths list
api.SelectSignals(f1, pathsList1)
api.SelectSignals(f2, pathsList2)
# Might use also
# QVizSelectSignals(f1.dataTreeView, pathsList1).execute()
# QVizSelectSignals(f2.dataTreeView, pathsList2).execute()

# Show the data tree view window
f1.show()
f2.show()

# Plot signal nodes
# Note: Data tree view does not need to be shown in order for this routine to
#       work
api.PlotSelectedSignalsFrom(f)


# Plot data from the first data source (f1) to Table Plot View
QVizSignalHandling(f1.dataTreeView).onPlotToTablePlotView(all_DTV=False)

# Plot data from the first data source (f1) to Stacked Plot View
QVizSignalHandling(f1.dataTreeView).onPlotToStackedPlotView(all_DTV=False)

# Keep the application running
sys.exit(app.exec_())
```

2.7.4.3 Example 2b

```
#!/usr/bin/python
"""This example demonstrates the procedure of plotting multiple arrays from
two IMAS IDS databases to a single plot.
"""

# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
```

(continues on next page)

(continued from previous page)

```
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.util.GlobalOperations import GlobalOperations
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource import DataSourceFactory
from imasviz.VizUtils.QVizGlobalValues import QVizGlobalValues

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
GlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = DataSourceFactory()

# Set and empty list for listing data tree view frames
f = []
# Set list of shots
n_shot = [52702, 52703]

for i in range(0, 2):
    # Load IMAS databases
    dataSource = dataSourceFactory.create(dataSourceName=QVizGlobalValues.IMAS_NATIVE,
                                          shotNumber=n_shot[i],
                                          runNumber=0,
                                          userName='imas_public',
                                          imasDbName='west')
    # Append data tree view frame to list
    f.append(api.CreateDataTree(dataSource))

# Set the list of node paths (for both databases) that are to be selected
paths1 = []
for i in range(1, 3):
    paths1.append('magnetics/flux_loop(' + str(i) + ')/flux/data')
paths2 = []
for i in range(1, 3):
    paths2.append('magnetics/bpol_probe(' + str(i) + ')/field/data')

# Select signal nodes corresponding to the paths in paths list
api.SelectSignals(f[0], paths1)
api.SelectSignals(f[1], paths2)
# Plot signal nodes
# Note: Data tree view does not need to be shown in order for this routine to
#       work
api.PlotSelectedSignalsFrom(f)

# Show the data tree view window
f[0].show()
f[1].show()

# Keep the application running
```

(continues on next page)

(continued from previous page)

```
app.exec()
```

2.7.4.4 Example 3

```
#!/usr/bin/python
"""This example demonstrates the procedure of plotting multiple arrays to a
single Table Plot View.
"""

# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizUtils.QVizGlobalValues import QVizGlobalValues

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
QVizGlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = QVizDataSourceFactory()

# Load IMAS database
dataSource = dataSourceFactory.create(
                                    dataSourceName=QVizGlobalValues.IMAS_NATIVE,
                                    shotNumber=52682,
                                    runNumber=0,
                                    userName='imas_public',
                                    imasDbName='west')

# Database on the GateWay HPC (comment the above dataSource code and uncomment
# the one below)
# dataSource = dataSourceFactory.create(shotNumber=52344,
#                                       runNumber=0,
#                                       userName='g2penkod',
#                                       imasDbName='viztest')

# Build the data tree view frame
f = api.CreateDataTree(dataSource)

# Set the list of node paths that are to be selected
pathsList = []
for i in range(0, 5):
    pathsList.append('magnetics/flux_loop(' + str(i) + ')/flux/data')
```

(continues on next page)

(continued from previous page)

```

# Select signal nodes corresponding to the paths in paths list
api.SelectSignals(f, pathsList)

# Plot the set of signal nodes selected by the user to a new Table Plot View.
api.PlotSelectedSignalsInTablePlotViewFrame(f)

# Show the data tree view window
api.ShowDataTree(f)

# Keep the application running
app.exec()
```

2.7.4.5 Example 4

```
#!/usr/bin/python

import os
# A module providing a number of functions and variables that can be used to
# manipulate different parts of the Python runtime environment.
import sys
# PyQt library imports
from PyQt5.QtWidgets import QApplication
# IMASViz source imports
from imasviz.VizUtils.QVizGlobalOperations import QVizGlobalOperations
from imasviz.Viz_API import Viz_API
from imasviz.VizDataSource.QVizDataSourceFactory import QVizDataSourceFactory
from imasviz.VizGUI.VizGUICommands.VizDataSelection.QVizSelectSignals import␣
↪QVizSelectSignals
from imasviz.VizGUI.VizGUICommands.VizDataSelection.QVizUnselectAllSignals import␣
↪QVizUnselectAllSignals
from imasviz.VizUtils.QVizGlobalValues import QVizGlobalValues

# Set object managing the PyQt GUI application's control flow and main
# settings
app = QApplication(sys.argv)

# Check if necessary system variables are set
QVizGlobalOperations.checkEnvSettings()

# Set Application Program Interface
api = Viz_API()

# Set data source retriever/factory
dataSourceFactory = QVizDataSourceFactory()

# Set user (get current user)
userName = os.environ['USER']

# Load IMAS database
dataSource = dataSourceFactory.create(
                                    dataSourceName=QVizGlobalValues.IMAS_NATIVE,
                                    shotNumber=52344,
                                    runNumber=0,
                                    userName='imas_public',
```

(continues on next page)

(continued from previous page)

```
                                    imasDbName='west')

# Database on the GateWay HPC (comment the above dataSource code and uncomment
# the one below)
# dataSource = dataSourceFactory.create(shotNumber=52344,
#                                       runNumber=0,
#                                       userName='g2penkod',
#                                       imasDbName='viztest')

# Build the data tree view frame
f = api.CreateDataTree(dataSource)

# Set configuration file
configFilePath = os.environ['HOME'] + "/.imasviz/configuration_name.pcfg"

# Extract signal paths from the config file and add them to a list of
# paths
pathsMap = QVizGlobalOperations.getSignalsPathsFromConfigurationFile(
    configFile=configFilePath)

# First unselect all signals (optional)
# QVizUnselectAllSignals(dataTreeView=f.dataTreeView).execute()

# Select the signals, defined by a path in a list of paths, in the
# given Data Tree View (DTV) window
QVizSelectSignals(dataTreeView=f.dataTreeView,
                  pathsMap=pathsMap).execute()

# Plot the set of the signal nodes selected using plot configuration file to
# a new Table Plot View and apply plot configurations (colors, line width etc.)
api.ApplyTablePlotViewConfiguration(f, configFilePath=configFilePath)

# Keep the application running
app.exec()
```

CHAPTER

THREE

HOWTOS

3.1 Developing a custom user interface (UI) plugins with Qt designer

Qt Designer is a tool for designing and building **Qt-based graphical user interfaces**. It allows the user to design custom widgets, dialogs, main windows, etc. using on-screen forms and a user-friendly simple drag-and-drop interface. It also provides the user with a convenient ability to preview the designs to ensure they work as intended.

In general, Qt Designer mainly offers basic Qt widgets such as *Push Button*, *Line Edit*, *List Widget*, etc. This list of the **Qt Designer** widgets can be extended by writing so-called **Qt designer plugins** (do not confuse with **IMASViz plugins**!). Normally this is done using C++ but PyQt5 also allows you to write Qt Designer plugins in Python3 programming language.

Such designer plugin is used to pass a custom widget source code (written in Python3) to Qt Designer. This way the widget becomes available within the Qt designer where it can be interactively moved, designed, connected to signals and slots and more.

Note: For more information on Qt Designer and PyQt5 based plugins and widgets check this link.

In this HOWTO section it will be described how to:

1. Develop a **custom PyQt5 widget**
2. Pass the **custom PyQt5 widget** class to **Qt designer** as a **Qt designer plugin**
3. Use the **custom PyQt5 widget** as a **Qt designer plugin** within the Qt Designer
4. Design of a custom **user interface (UI) plugin** (which includes the custom **Qt designer plugin**) with Qt designer
5. Use the **UI plugin** in a standalone way as a **PyQt5 application**
6. Use the **UI plugin** in **IMASViz**

Warning: Qt version of used PyQt5 (compiled with Qt) and Qt designer must match!

For the purposes of this HowTo section, a widget source code for the **Magnetics IDS overview Plugin** was developed and it is available in the IMASViz source code (VizPlugins/viz_example). As it is mainly intended only as an example of a plugin (including an example of the widget source code), it is referred to mainly as an **Example Plugin** (same goes for source files - exampleWidget.py and exampleplugin.py, introduced later in this howTo section).

As and addition, below is a short demonstration video of **SOLPS overview Plugin**, showing an example of the processes listed in points **3-6**. More on this plugin (as IMASViz plugin) can be found in section *Plugins*.

Local Video Link.

3.1.1 Custom PyQt5 widget creation (code development)

This section describes and demonstrates how to write a complete custom PyQt5 widget that handles data stored within the **Magnetics IDS**. Later in this HowTo section, the same widget will be then used to create **Magnetics IDS overview Plugin** using Qt designer.

The main final features of this custom plugin will be:

- – **Opening and reading the contents a specified IDS** (specified by the set case parameters)

  OR

  – **reading the contents of an IDS which was passed to the plugin by "host" application**, and
- convenient plotting of all **flux_loop** and **poloidal field probe** (AoS) signals found in the **Magnetics IDS** (arrays of values).

In this case, the whole widget source code is written in Python3 file named **exampleWidget.py**.

The final code can be already observed and compared here: *Full final code of the example PyQt5 widget*.

Note: It is highly recommended to have the finished code opened on the side while going through this HowTo section to better understand the whole code.

Note: It is recommended to have at least basic knowledge from programming (especially with Python programming language) before proceeding with the widget development instructions. A complete beginner might find those instructions a bit overwhelming.

The steps are split into the next subsections:

1. Code header
2. Import statements
3. Widget Class definition
   - 3.1 Widget Constructor definition
   - 3.2 Widget base "set" and "get" routines
   - 3.3 Widget custom routines
4. PlotCanvas Class definition
   - 4.1 PlotCanvas Constructor definition
   - 4.2 PlotCanvas custom plotting routines
5. Running the code as the main program (standalone)

3.1.1.1 Code header

The header of the custom PyQt5 widget source code should contain the basic information about the code itself: - the source code (.py) filename, - short description and the purpose of this script - authors name and - authors contact (e-mail is most convenient).

```
#  Name    : exampleWidget
#
#           A PyQt5 widget to serve as an example of how to create custom widgets
#           that can be used within Qt designer.
#           This widget embeds Matplotlib canvas (plot space). It contains also
#           defined PyQt5 slots for setting the magnetics IDS parameters and
#           a slot (function) for executing the plot procedure,
#           populating/filling the Matplotlib canvas.
#
#  Author :
#          Dejan Penko
#  E-mail :
#          dejan.penko@lecad.fs.uni-lj.si
#
#*****************************************************
#     Copyright(c) 2019- D. Penko
```

Documentation should be as important to a developer as all other facets of development. **Every code should include documentation** (in the forms of a header, code comments, etc.). It either explains what the code does, how it operates, how to use it etc. Documentation is an important part of software engineering.

No matter what the code contains, chances are that someday other users will try to understand and use it. Taking that extra time to write a proper description of the contents of the code will save huge amounts of time and effort for everybody to understand the code.

3.1.1.2 Import statements

The custom PyQt5 widget requires additional sources - modules.

The ones required in this case are:

- The common system, OS and logging modules:

```
# import module providing system-specific parameters and functions
import sys
# import module providing miscellaneous operating system interfaces
import os
# import module providing log handling
import logging
```

- PyQt5 modules:

```
# import modules providing PyQt5 parameters, functions etc.
from PyQt5.QtWidgets import QApplication, QWidget, QMainWindow, QVBoxLayout,␣
↪QSizePolicy
from PyQt5.QtCore import pyqtSlot, pyqtSignal
```

- Matplotlib modules and setting matplotlib to use the Qt rendering:

```
# import module providing matplotlib parameters, functions etc.
import matplotlib
matplotlib.use('Qt5Agg') # Use Qt rendering
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as␣
↪NavigationToolbar
```

- IMAS modules:

```
33 # import module providing IMAS and IDS-related parameters, functions etc.
34 import imas
```

3.1.1.3 Widget class

This section describes and demonstrates how to define a new widget class in Python3.

Class definition

The initial and important part of this code is the definition of a new class inheriting from the PyQt5 **QWidget** class. In this case, the class is named *exampleWidget*.

This class will later fully define the QWidget (contents, design, functions related to the widget and more).

```
36 class exampleWidget(QWidget):
37     """A widget for opening magnetics IDS, extracting the flux loop or
38     poloidal probe quantities and plotting them to matplotlib figure canvas.
39     """
```

Note: Do not forget to describe the class - what is its purpose etc.

Here also a new PyQt signal is set, which will be needed later in code.

```
41     # Create a custom signal
42     idsSet = pyqtSignal(bool)
```

In short, the signal on its own does not perform any action. Instead, it is `connected` to a `slot`. The slot can be any callable Python function. When the signal gets emitted the connected slot (Python function) gets called.

Note: More on signal and slots: Link

Constructor definition

In short, constructors are generally used for initiating an object. The task of constructors is to initialize the data members of the class when an object of the class is created.

In the case of this custom widget, the constructor required two additional arguments:

- *parent* (can be either Qt object, later our case QMainWindow), and
- *ids* (an IDS object).

Both arguments are set as **None** (default values).

```
44     def __init__(self, parent=None, ids=None, *args, **kwargs):
45         """
46         Arguments:
47             parent (PyQt5 object) : Qt widget parent (e.g. QMainWindow)
48             ids    (IDS object)   : IDS object - optional parameter.
49                                     This widget does not require IDS object in
50                                     order to work (the default IDS parameters
```

(continues on next page)

(continued from previous page)

```
                                    will be used to open the IDS).
                                    However, if an IDS object is already
                                    available it can be passed to the widget
                                    to be used instead (for example, passing
                                    the IDS object from the IMASViz to this
                                    widget).
        """
```

And the *ids* object is set with:

```
        # Set IDS object
        # Note: if not provided as an argument it will be set to None
        self.ids = ids
```

Regarding the *ids* object, the main idea is to make our widget capable of performing in two different ways. Either:

- use IDS object passed to the widget (in which case **ids != None**),
- if no IDS object was passed (**ids == None**), open IDS and create a new *ids* object.

For example, in **IMASViz** there is at least one IDS open at the time and thus have the *ids* object available. Instead of opening the IDS again, the *ids* object can be just passed to the custom widget as an argument and the widget can continue to use it.

If there is no IDS object available (meaning no IDS is already being opened), an IDS must be opened thus creating an object referring to the IDS. In the constructor we define a dictionary labeled as *self.idsParameters* which should contain all IDS parameters for IDS (will be used later to open the needed IDS):

```
        # Set IDS case parameters
        # - Empty dictionary
        self.idsParameters = {}
        # - shot
        self.idsParameters['shot'] = '52344'
        # - run r
        self.idsParameters['run'] = '0'
        # - user
        self.idsParameters['user'] = os.getenv('USER')
        # - device / machine / database name
        self.idsParameters['device'] = 'viztest'
        # - IMAS major version (3.x.y)
        self.idsParameters['IMAS major version'] = '3'
        # - label of the IDS to be used
        self.idsParameters['idsName'] = 'magnetics'
```

Constructor should contain also a check if the widget is being run in a desktop environment. This is mandatory as this is a widget which deals with GUI and visualization. The code should not be run from a "terminal-only" environment (for example `ssh user@host` etc.).

In this case we define a function named *checkDisplay()*:

```
    @pyqtSlot()
    def checkDisplay(self):
        try:
            os.environ['DISPLAY']
        except:
            logging.error('No display available!')
```

and execute it in the constructor:

```
        # Check if display is available (display is mandatory, as this is
        # PyQt5 widget)
        self.checkDisplay()
```

Lastly, a widget layout and its contents need to be defined (plot canvas and matplotlib navigation toolbar):

Note: The **PlotCanvas** class definition and the definition of its routines will be done in the following sections.

```
        # Set widget layout
        self.setLayout(QVBoxLayout())
        # Set empty matplotlib canvas
        self.canvas = PlotCanvas(self)
        # Set matplotlib toolbar
        self.toolbar = NavigationToolbar(self.canvas, self)
        # Add canvas and toolbar to widget layout
        self.layout().addWidget(self.canvas)
        self.layout().addWidget(self.toolbar)
```

Base "set" and "get" routines

For setting and getting/returning the IDS case parameters, the definition of a few *set/get* routines is required.

The *set* routines must be set as **slots (@pyqtSlot)**. This clearly marks the function as a slot for PyQt5 and it also increases its speed and performance when being executed as slots in PyQt5 applications.

The *get* routines are simple functions which return one variable value.

```
    @pyqtSlot(str)
    def setShot(self, shot):
        self.idsParameters['shot'] = shot

    def getShot(self):
        return self.idsParameters['shot']

    @pyqtSlot(str)
    def setRun(self, run):
        self.idsParameters['run'] = run

    def getRun(self):
        return self.idsParameters['run']

    @pyqtSlot(str)
    def setUser(self, user):
        self.idsParameters['user'] = user

    def getUser(self):
        return self.idsParameters['user']

    @pyqtSlot(str)
    def setDevice(self, device):
        self.idsParameters['device'] = device

    def getDevice(self):
        return self.idsParameters['device']
```

(continues on next page)

(continued from previous page)

```
173
174     @pyqtSlot(str)
175     def setIMASmVer(self, ver):
176         self.idsParameters['IMAS major version'] = ver
177
178     def getIMASmVer(self):
179         return self.idsParameters['IMAS major version']
180
181     @pyqtSlot(str)
182     def setIDSname(self, idsName):
183         self.idsParameters['idsName'] = idsName
184
185     def getIDSname(self):
186         return self.idsParameters['idsName']
```

Custom functions (routines)

The first "bundle" of functions deals with IDSs:

1. *openIDS*: for opening the IDS (using the IDS case parameters, defined with the *self.idsParameters* dictionary),

```
125     @pyqtSlot()
126     def openIDS(self):
127         """Open magnetics IDS.
128         """
129         # Open IDS
130         self.ids = imas.ids(int(self.idsParameters['shot']),
131                             int(self.idsParameters['run']))
132         self.ids.open_env(self.idsParameters['user'],
133                           self.idsParameters['device'],
134                           self.idsParameters['IMAS major version'])
135         # Get magnetics IDS
136         self.ids.magnetics.get()
```

2. *setIDS*: for setting the IDS object (*self.ids*). Here also the **emit signal** statement is included. This way every time this function will be called/executed, this signal will get emmited. Later in the plugin this signal will be used to initiate the execution of certain functions on signal-emit.

```
138     def setIDS(self, ids):
139         self.ids = ids
140         # Emit signal indicating that the IDS object is set
141         self.idsSet.emit(False)
```

3. *getIDS*: for getting/returning the IDS object (*self.ids*).

```
143     def getIDS(self):
144         return self.ids
```

The second "bundle" of functions deals with executing the plotting procedures to populate the **matplotlib canvas**. At this point in this tutorial, the **PlotCanvas** class is not yet defined. This will be done in the next HowTo section. The functions needed are:

- *plotFluxAoS*: for plotting all **Flux_loop** signal arrays values, and
- *plotBPolAoS*: for plotting all **poloidal field probe** signal arrays values

```
95      @pyqtSlot()
96      def plotFluxAoS(self):
97          """Plot Flux Loop arrays to canvas.
98          """
99
100         # IDS check
101         if self.ids == None:
102             logging.error(' IDS was not set/opened!')
103             return
104         # Canvas figure check
105         if self.canvas.figure != None:
106             self.canvas.figure.clear()
107         # Plot flux loop Aos
108         self.canvas.plotFluxAoS(self.ids)
109
110     @pyqtSlot()
111     def plotBPolAoS(self):
112         """Plot poloidal field probe arrays to canvas.
113         """
114
115         # IDS check
116         if self.ids == None:
117             logging.error(' IDS was not set/opened!')
118             return
119         # Canvas figure check
120         if self.canvas.figure != None:
121             self.canvas.figure.clear()
122         # Plot Poloidal field AoS
123         self.canvas.plotBPolAoS(self.ids)
```

3.1.1.4 PlotCanvas

This section describes and demonstrates how to define a new matplotlib FigureCanvas class in Python3.

Class definition

Second main part this code (with the first being the definition of the *exampleWindget* class) is the definition of a new class inheriting from the matplotlib *FigureCanvas* class. In this case, the class is named *PlotCanvas*.

This class will later fully define the matplotlib plot frame (canvas) and functions related to it.

```
195 class PlotCanvas(FigureCanvas):
196     """Matplotlib figure canvas that is to be embedded within the widget.
197     FigureCanvas is the area onto which the figure is drawn
198     """
```

Constructor definition

In this case the constructor takes 4 additional arguments:

- *parent* (our custom QWidget),
- *width* (canvas width),
- *height* (canvas height), and

- *dpi* (dots per inch).

The *parent* argument is set as **None**, *width* to **5**, *height* to **5** and *dpi* to **100** (default values).

```
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        """
        Arguments:
            parent (PyQt5 object) : PyQt5 parent (e.g. QWidget).
            width  (int)          : Canvas width.
            height (int)          : Canvas height.
            dpi    (int)          : Dots per inch.
        """
```

Next, a figure object *fig* is set:

Note: **Figure** routine is taken from the import statement (see *Import statements*).

```

    @pyqtSlot()
    def plotBPolAoS(self):
        """Plot poloidal field probe arrays to canvas.
        """

        # IDS check
        if self.ids == None:
            logging.error(' IDS was not set/opened!')
            return
        # Canvas figure check
        if self.canvas.figure != None:
            self.canvas.figure.clear()
        # Plot Poloidal field AoS
        self.canvas.plotBPolAoS(self.ids)

    @pyqtSlot()
    def openIDS(self):
        """Open magnetics IDS.
        """
        # Open IDS
        self.ids = imas.ids(int(self.idsParameters['shot']),
                            int(self.idsParameters['run']))
        self.ids.open_env(self.idsParameters['user'],
                          self.idsParameters['device'],
                          self.idsParameters['IMAS major version'])
        # Get magnetics IDS
        self.ids.magnetics.get()

    def setIDS(self, ids):
        self.ids = ids
        # Emit signal indicating that the IDS object is set
        self.idsSet.emit(False)

    def getIDS(self):
        return self.ids

    @pyqtSlot(str)
    def setShot(self, shot):
        self.idsParameters['shot'] = shot
```

(continues on next page)

(continued from previous page)

```
149
150     def getShot(self):
151         return self.idsParameters['shot']
152
153     @pyqtSlot(str)
154     def setRun(self, run):
155         self.idsParameters['run'] = run
156
157     def getRun(self):
158         return self.idsParameters['run']
159
160     @pyqtSlot(str)
161     def setUser(self, user):
162         self.idsParameters['user'] = user
163
164     def getUser(self):
165         return self.idsParameters['user']
166
167     @pyqtSlot(str)
168     def setDevice(self, device):
169         self.idsParameters['device'] = device
170
171     def getDevice(self):
172         return self.idsParameters['device']
173
174     @pyqtSlot(str)
175     def setIMASmVer(self, ver):
176         self.idsParameters['IMAS major version'] = ver
177
178     def getIMASmVer(self):
179         return self.idsParameters['IMAS major version']
180
181     @pyqtSlot(str)
182     def setIDSname(self, idsName):
183         self.idsParameters['idsName'] = idsName
184
185     def getIDSname(self):
186         return self.idsParameters['idsName']
187
188     @pyqtSlot()
189     def checkDisplay(self):
190         try:
191             os.environ['DISPLAY']
192         except:
193             logging.error('No display available!')
194
195 class PlotCanvas(FigureCanvas):
196     """Matplotlib figure canvas that is to be embedded within the widget.
197     FigureCanvas is the area onto which the figure is drawn
198     """
199
200     def __init__(self, parent=None, width=5, height=4, dpi=100):
201         """
202         Arguments:
203             parent (PyQt5 object) : PyQt5 parent (e.g. QWidget).
204             width  (int)          : Canvas width.
205             height (int)          : Canvas height.
```

(continues on next page)

(continued from previous page)

```
            dpi    (int)          : Dots per inch.
        """

        # Set figure
        fig = Figure(figsize=(width, height), dpi=dpi)
```

Next, the (by class) inherited *FigureCanvas* constructor is executed. The *fig* object is passed to it as an argument. This way the **figure** is embedded within the **matplotlib canvas**.

```
        # Set canvas (pass figure)
        FigureCanvas.__init__(self, fig)
```

Next, the **parent** of the *FigureCanvas* is set:

```
        # Set canvas parent
        self.setParent(parent)
```

Lastly, the *FigureCanvas* **size policy** is set.

```
        # Set canvas size policy
        FigureCanvas.setSizePolicy(self,
                                   QSizePolicy.Expanding,
                                   QSizePolicy.Expanding)
```

The whole *PlotCanvas* constructor code:

```
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        """
        Arguments:
            parent (PyQt5 object) : PyQt5 parent (e.g. QWidget).
            width  (int)          : Canvas width.
            height (int)          : Canvas height.
            dpi    (int)          : Dots per inch.
        """

        # Set figure
        fig = Figure(figsize=(width, height), dpi=dpi)
        # Set canvas (pass figure)
        FigureCanvas.__init__(self, fig)
        # Set canvas parent
        self.setParent(parent)
        # Set canvas size policy
        FigureCanvas.setSizePolicy(self,
                                   QSizePolicy.Expanding,
                                   QSizePolicy.Expanding)
```

Custom plotting functions

There are two plotting functions required:

1. *plotFluxAoS*, for plotting all **flux_loop** signal arrays, and
2. *plotBPolAoS*, for plotting all **poloidal field probe** signal arrays values.

Both are very similar, the only difference between them is which data is extracted from the IDS and used for plotting. Because of this similarity, only the function *plotFluxAoS* will be described in depth.

The function *plotFluxAoS* requires only one argument: the IDS object. The function must also set the provided *ids* object to *self.ids* object.

```
    def plotFluxAoS(self, ids):
        """Plot values found in flux loops AoS.

        Arguments:
            ids (IDS object) : IDS object referring to the IDS from which the
                               data is to be extracted.
        """

        # Add/Update IDS reference to the object (figure canvas)
        self.ids = ids
        # IDS check
        if self.ids == None:
            logging.error('IDS was not set/opened!')
            return
```

Next, figure subplot must be set:

```
        # Set subplot
        ax = self.figure.add_subplot(111)
```

Next, the time values must be extracted and assigned to *time_values* array. The time values will correspond to plot X-axis, thus, for easier representation, a new array *x* can be defined and the same values assigned to it.

```
        # Extract X-axis values (time)
        time_values = self.ids.magnetics.time
        x = time_values
```

Next, looping through all structured of the **flux_loop** AoS is required. Each array values (Y-axis values) need to be extracted and then together with the previously set time values (X-axis) a new plot can be added to the matplotlib figure. Because of the loop, this gets repeated until no more AoS arrays are left.

```
        # Get the size of AoS (number of arrays)
        num_flux_loop_AoS = len(self.ids.magnetics.flux_loop)
        # For each array extract array values and create a plot
        for i in range(num_flux_loop_AoS):
            # Extract array values
            y = self.ids.magnetics.flux_loop[i].flux.data
            # Set plot (line) defined by X and Y values +
            # set line as full line (-) and add legend label.
            ax.plot(x, y, '-', label='Flux_loop[' + str(i) + ']')
```

Next, few additional modifications are required:

- enabling plot grid,
- setting X-axis, Y-axis label,
- setting plot title,
- enabling legend, and
- drawing the plot.

```
        # Enable grid
        ax.grid()
        # Set axis labels and plot title
```

(continues on next page)

(continued from previous page)

```
        ax.set(xlabel='time [s]', ylabel='Flux Loop values',
               title='Flux loop')
        # Enable legend
        ax.legend()
        # Draw/Show plots
        self.draw()
```

Final *plotFluxAoS* code:

```
    def plotFluxAoS(self, ids):
        """Plot values found in flux loops AoS.

        Arguments:
            ids (IDS object) : IDS object referring to the IDS from which the
                               data is to be extracted.
        """

        # Add/Update IDS reference to the object (figure canvas)
        self.ids = ids
        # IDS check
        if self.ids == None:
            logging.error('IDS was not set/opened!')
            return
        # Set subplot
        ax = self.figure.add_subplot(111)
        # Extract X-axis values (time)
        time_values = self.ids.magnetics.time
        x = time_values
        # Get the size of AoS (number of arrays)
        num_flux_loop_AoS = len(self.ids.magnetics.flux_loop)
        # For each array extract array values and create a plot
        for i in range(num_flux_loop_AoS):
            # Extract array values
            y = self.ids.magnetics.flux_loop[i].flux.data
            # Set plot (line) defined by X and Y values +
            # set line as full line (-) and add legend label.
            ax.plot(x, y, '-', label='Flux_loop[' + str(i) + ']')
        # Enable grid
        ax.grid()
        # Set axis labels and plot title
        ax.set(xlabel='time [s]', ylabel='Flux Loop values',
               title='Flux loop')
        # Enable legend
        ax.legend()
        # Draw/Show plots
        self.draw()
```

As already stated, *plotBPolAoS* function code is almost identical to *plotFluxAoS* code.

Final *plotBPolAoS*:

```
    def plotBPolAoS(self, ids):
        """Plot poloidal field probe values.

        Arguments:
            ids (IDS object) : IDS object referring to the IDS from which the
                               data is to be extracted.
```

(continues on next page)

(continued from previous page)

```
        """
        # Add/Update IDS reference to the object (figure canvas)
        self.ids = ids
        # IDS check
        if self.ids == None:
            logging.error('IDS was not set/opened!')
            return
        # Set subplot
        ax = self.figure.add_subplot(111)
        # Extract X-axis values (time)
        time_values = self.ids.magnetics.time
        x = time_values
        # Get the size of AoS (number of arrays)
        num_bpol_probe_AoS = len(self.ids.magnetics.bpol_probe)
        # For each array extract array values and create a plot
        for i in range(num_bpol_probe_AoS):
            # Extract array values
            y = self.ids.magnetics.bpol_probe[i].field.data
            # Set plot (line) defined by X and Y values +
            # set line as full line (-) and add legend label.
            ax.plot(x, y, '-', label='bpol_probe[' + str(i) + ']')
        # Enable grid
        ax.grid()
        # Set axis labels and plot title
        ax.set(xlabel='time [s]', ylabel='Poloidal field probe values',
               title='Poloidal field probe')
        # Enable legend
        ax.legend()
        # Draw/Show plots
        self.draw()
```

At this point the **exampleWidget.py** code is finished and ready for use.

3.1.1.5 Running the code as the main program (standalone)

To run this example widget in a standalone way, few more lines must be added to the **exampleWidget.py**.

This part of the code contains setting the *QApplication*, *QMainWindow*, initiating the *exampleWidget* class, reading the IDS (parameters are set in the *exampleWidget* constructor) and executing the plotting procedures.

```
if __name__ == '__main__':

    # Set application object
    app = QApplication(sys.argv)
    # Set main PyQt5 window
    mainWindow = QMainWindow()
    # Set window title
    mainWindow.setWindowTitle('Example Widget')
    # Set example widget object
    ew = exampleWidget()
    # Open IDS (magnetics IDS)
    ew.openIDS()
    # Plot Flux Loop arrays
    ew.plotFluxAoS()
    # Plot poloidal field probe arrays (an option other than plotFluxAoS)
    # ew.plotBPolAoS()
```

(continues on next page)

(continued from previous page)

```
    # Set example widget as a central widget of the main window
    mainWindow.setCentralWidget(ew)
    # Show the main window
    mainWindow.show()
    # Keep the application running (until the 'exit application' command is
    # executed
    sys.exit(app.exec_())
```

The code can now be run from the terminal with the next command:

```
python3 exampleWidget.py
```

Note: Make sure that the IDS with the specified case parameters exists (done in *Constructor definition* using *idsParameters* dictionary)!

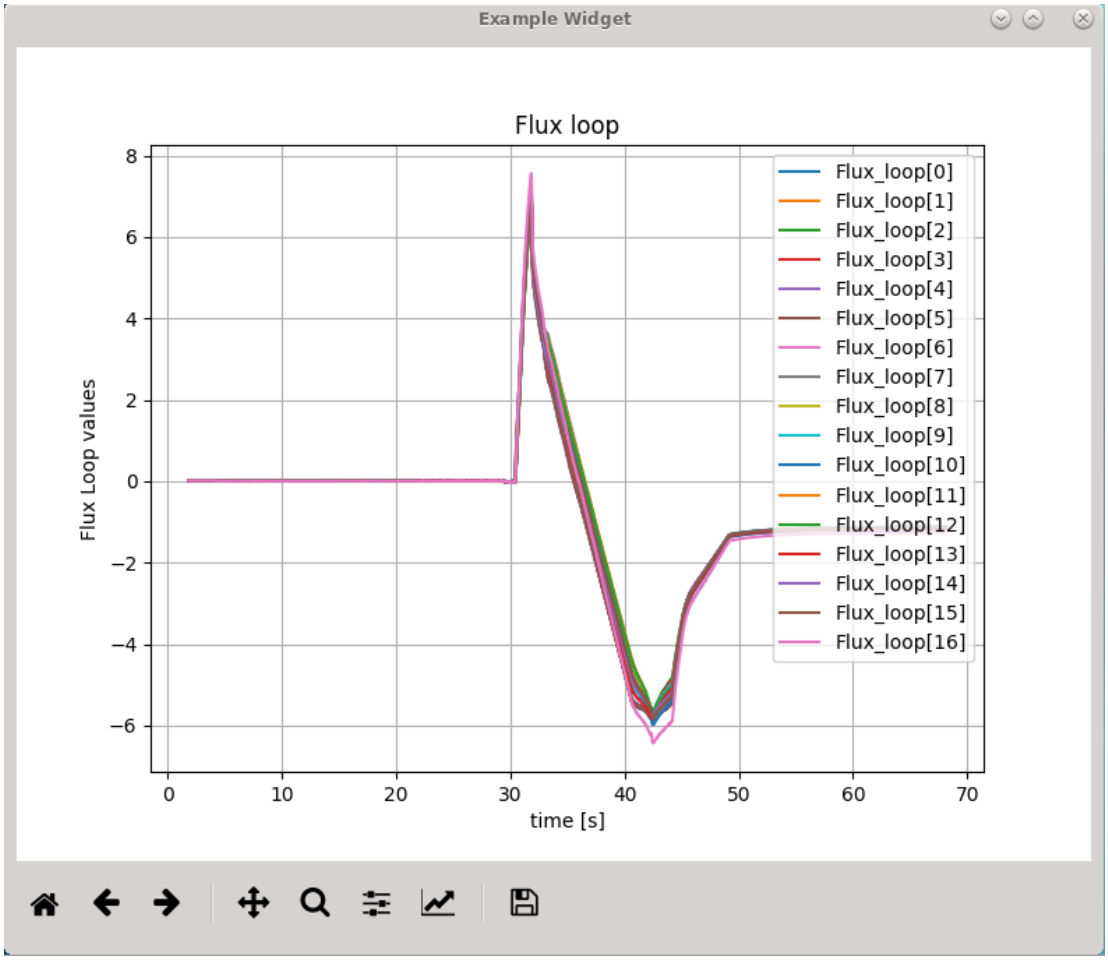


Fig. 1: **exampleWidget.py**: Plotting all **magnetics IDS** arrays (17) of AoS **flux_loop** found in IDS (on GateWay HPC) shot: 52344; run: 0; user: g2penkod; device: viztest.

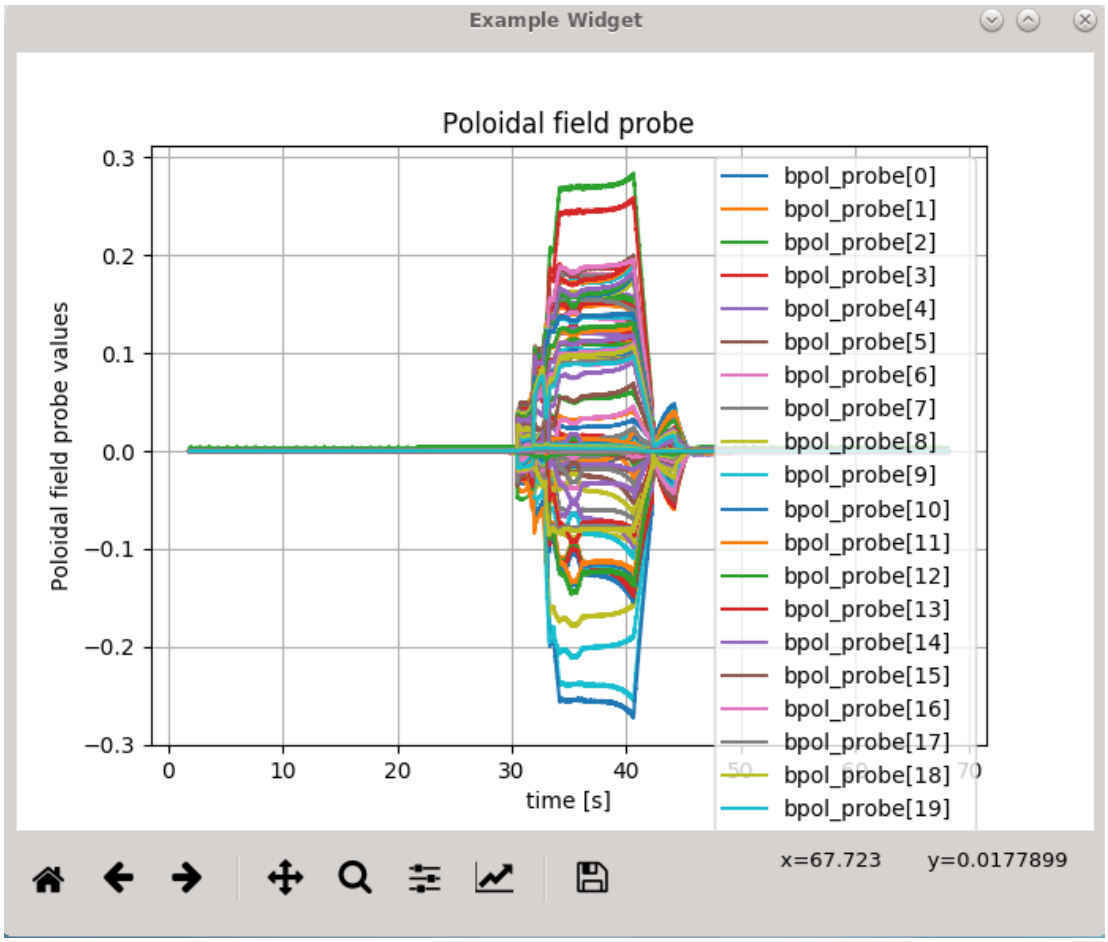


Fig. 2: **exampleWidget.py**: Plotting all **magnetics IDS** arrays (ABOUT 130) of AoS **bpol_probe** found in IDS (on GateWay HPC) shot: 52344; run: 0; user: g2penkod; device: viztest.

3.1.1.6 Full final code of the example PyQt5 widget

```
#  Name    : exampleWidget
#
#           A PyQt5 widget to serve as an example of how to create custom widgets
#           that can be used within Qt designer.
#           This widget embeds Matplotlib canvas (plot space). It contains also
#           defined PyQt5 slots for setting the magnetics IDS parameters and
#           a slot (function) for executing the plot procedure,
#           populating/filling the Matplotlib canvas.
#
#  Author :
#          Dejan Penko
#  E-mail :
#          dejan.penko@lecad.fs.uni-lj.si
#
#*****************************************************
#      Copyright(c) 2019- D. Penko

# import module providing system-specific parameters and functions
```

(continues on next page)

(continued from previous page)

```
import sys
# import module providing miscellaneous operating system interfaces
import os
# import module providing log handling
import logging
# import modules providing PyQt5 parameters, functions etc.
from PyQt5.QtWidgets import QApplication, QWidget, QMainWindow, QVBoxLayout,␣
↪QSizePolicy
from PyQt5.QtCore import pyqtSlot, pyqtSignal
# import module providing matplotlib parameters, functions etc.
import matplotlib
matplotlib.use('Qt5Agg') # Use Qt rendering
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as␣
↪NavigationToolbar
# import module providing IMAS and IDS-related parameters, functions etc.
import imas

class exampleWidget(QWidget):
    """A widget for opening magnetics IDS, extracting the flux loop or
    poloidal probe quantities and plotting them to matplotlib figure canvas.
    """

    # Create a custom signal
    idsSet = pyqtSignal(bool)

    def __init__(self, parent=None, ids=None, *args, **kwargs):
        """
        Arguments:
            parent (PyQt5 object) : Qt widget parent (e.g. QMainWindow)
            ids    (IDS object)   : IDS object - optional parameter.
                                    This widget does not require IDS object in
                                    order to work (the default IDS parameters
                                    will be used to open the IDS).
                                    However, if an IDS object is already
                                    available it can be passed to the widget
                                    to be used instead (for example, passing
                                    the IDS object from the IMASViz to this
                                    widget).
        """
        # Run QWidget constructor
        super(QWidget, self).__init__(parent)

        # Check if display is available (display is mandatory, as this is
        # PyQt5 widget)
        self.checkDisplay()

        # Set IDS object
        # Note: if not provided as an argument it will be set to None
        self.ids = ids

        # Set IDS case parameters
        # - Empty dictionary
        self.idsParameters = {}
        # - shot
        self.idsParameters['shot'] = '52344'
```

(continues on next page)

(continued from previous page)

```
        # - run r
        self.idsParameters['run'] = '0'
        # - user
        self.idsParameters['user'] = os.getenv('USER')
        # - device / machine / database name
        self.idsParameters['device'] = 'viztest'
        # - IMAS major version (3.x.y)
        self.idsParameters['IMAS major version'] = '3'
        # - label of the IDS to be used
        self.idsParameters['idsName'] = 'magnetics'

        # Set widget layout
        self.setLayout(QVBoxLayout())
        # Set empty matplotlib canvas
        self.canvas = PlotCanvas(self)
        # Set matplotlib toolbar
        self.toolbar = NavigationToolbar(self.canvas, self)
        # Add canvas and toolbar to widget layout
        self.layout().addWidget(self.canvas)
        self.layout().addWidget(self.toolbar)

    @pyqtSlot()
    def plotFluxAoS(self):
        """Plot Flux Loop arrays to canvas.
        """

        # IDS check
        if self.ids == None:
            logging.error(' IDS was not set/opened!')
            return
        # Canvas figure check
        if self.canvas.figure != None:
            self.canvas.figure.clear()
        # Plot flux loop Aos
        self.canvas.plotFluxAoS(self.ids)

    @pyqtSlot()
    def plotBPolAoS(self):
        """Plot poloidal field probe arrays to canvas.
        """

        # IDS check
        if self.ids == None:
            logging.error(' IDS was not set/opened!')
            return
        # Canvas figure check
        if self.canvas.figure != None:
            self.canvas.figure.clear()
        # Plot Poloidal field AoS
        self.canvas.plotBPolAoS(self.ids)

    @pyqtSlot()
    def openIDS(self):
        """Open magnetics IDS.
        """
        # Open IDS
        self.ids = imas.ids(int(self.idsParameters['shot']),
```

(continues on next page)

(continued from previous page)

```
                            int(self.idsParameters['run']))
        self.ids.open_env(self.idsParameters['user'],
                          self.idsParameters['device'],
                          self.idsParameters['IMAS major version'])
        # Get magnetics IDS
        self.ids.magnetics.get()

    def setIDS(self, ids):
        self.ids = ids
        # Emit signal indicating that the IDS object is set
        self.idsSet.emit(False)

    def getIDS(self):
        return self.ids

    @pyqtSlot(str)
    def setShot(self, shot):
        self.idsParameters['shot'] = shot

    def getShot(self):
        return self.idsParameters['shot']

    @pyqtSlot(str)
    def setRun(self, run):
        self.idsParameters['run'] = run

    def getRun(self):
        return self.idsParameters['run']

    @pyqtSlot(str)
    def setUser(self, user):
        self.idsParameters['user'] = user

    def getUser(self):
        return self.idsParameters['user']

    @pyqtSlot(str)
    def setDevice(self, device):
        self.idsParameters['device'] = device

    def getDevice(self):
        return self.idsParameters['device']

    @pyqtSlot(str)
    def setIMASmVer(self, ver):
        self.idsParameters['IMAS major version'] = ver

    def getIMASmVer(self):
        return self.idsParameters['IMAS major version']

    @pyqtSlot(str)
    def setIDSname(self, idsName):
        self.idsParameters['idsName'] = idsName

    def getIDSname(self):
        return self.idsParameters['idsName']

```

(continues on next page)

(continued from previous page)

```
    @pyqtSlot()
    def checkDisplay(self):
        try:
            os.environ['DISPLAY']
        except:
            logging.error('No display available!')

class PlotCanvas(FigureCanvas):
    """Matplotlib figure canvas that is to be embedded within the widget.
    FigureCanvas is the area onto which the figure is drawn
    """

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        """
        Arguments:
            parent (PyQt5 object) : PyQt5 parent (e.g. QWidget).
            width  (int)          : Canvas width.
            height (int)          : Canvas height.
            dpi    (int)          : Dots per inch.
        """

        # Set figure
        fig = Figure(figsize=(width, height), dpi=dpi)
        # Set canvas (pass figure)
        FigureCanvas.__init__(self, fig)
        # Set canvas parent
        self.setParent(parent)
        # Set canvas size policy
        FigureCanvas.setSizePolicy(self,
                                   QSizePolicy.Expanding,
                                   QSizePolicy.Expanding)

    def plotFluxAoS(self, ids):
        """Plot values found in flux loops AoS.

        Arguments:
            ids (IDS object) : IDS object referring to the IDS from which the
                               data is to be extracted.
        """

        # Add/Update IDS reference to the object (figure canvas)
        self.ids = ids
        # IDS check
        if self.ids == None:
            logging.error('IDS was not set/opened!')
            return
        # Set subplot
        ax = self.figure.add_subplot(111)
        # Extract X-axis values (time)
        time_values = self.ids.magnetics.time
        x = time_values
        # Get the size of AoS (number of arrays)
        num_flux_loop_AoS = len(self.ids.magnetics.flux_loop)
        # For each array extract array values and create a plot
        for i in range(num_flux_loop_AoS):
            # Extract array values
            y = self.ids.magnetics.flux_loop[i].flux.data
```

(continues on next page)

(continued from previous page)

```
            # Set plot (line) defined by X and Y values +
            # set line as full line (-) and add legend label.
            ax.plot(x, y, '-', label='Flux_loop[' + str(i) + ']')
        # Enable grid
        ax.grid()
        # Set axis labels and plot title
        ax.set(xlabel='time [s]', ylabel='Flux Loop values',
               title='Flux loop')
        # Enable legend
        ax.legend()
        # Draw/Show plots
        self.draw()

    def plotBPolAoS(self, ids):
        """Plot poloidal field probe values.

        Arguments:
            ids (IDS object) : IDS object referring to the IDS from which the
                               data is to be extracted.
        """
        # Add/Update IDS reference to the object (figure canvas)
        self.ids = ids
        # IDS check
        if self.ids == None:
            logging.error('IDS was not set/opened!')
            return
        # Set subplot
        ax = self.figure.add_subplot(111)
        # Extract X-axis values (time)
        time_values = self.ids.magnetics.time
        x = time_values
        # Get the size of AoS (number of arrays)
        num_bpol_probe_AoS = len(self.ids.magnetics.bpol_probe)
        # For each array extract array values and create a plot
        for i in range(num_bpol_probe_AoS):
            # Extract array values
            y = self.ids.magnetics.bpol_probe[i].field.data
            # Set plot (line) defined by X and Y values +
            # set line as full line (-) and add legend label.
            ax.plot(x, y, '-', label='bpol_probe[' + str(i) + ']')
        # Enable grid
        ax.grid()
        # Set axis labels and plot title
        ax.set(xlabel='time [s]', ylabel='Poloidal field probe values',
               title='Poloidal field probe')
        # Enable legend
        ax.legend()
        # Draw/Show plots
        self.draw()

if __name__ == '__main__':

    # Set application object
    app = QApplication(sys.argv)
    # Set main PyQt5 window
    mainWindow = QMainWindow()
    # Set window title
```

(continues on next page)

(continued from previous page)

```
    mainWindow.setWindowTitle('Example Widget')
    # Set example widget object
    ew = exampleWidget()
    # Open IDS (magnetics IDS)
    ew.openIDS()
    # Plot Flux Loop arrays
    ew.plotFluxAoS()
    # Plot poloidal field probe arrays (an option other than plotFluxAoS)
    # ew.plotBPolAoS()
    # Set example widget as a central widget of the main window
    mainWindow.setCentralWidget(ew)
    # Show the main window
    mainWindow.show()
    # Keep the application running (until the 'exit application' command is
    # executed
    sys.exit(app.exec_())
```

3.1.2 Passing custom PyQt5 widget to Qt designer

In order to pass the **custom PyQt5 widget** to **Qt designer**, a separate Qt plugin Python file is required, written in Python3 programming language. **The name of this file is of major importance** as if set improperly the Qt designer will not recognize it! The name of this file should end with **plugin.py** (**case sensitive!**). In this case, the file is named **exampleplugin.py**. it must be placed in the same directory as the widget source code - **exampleWidget.py**.

This plugin *.py* file for Qt designer follows a certain template which can be used and slightly modified as required..

The whole code is shown below.

```
#!/usr/bin/env python3

"""

An example of a widget as a Qt designer plugin. The widget plots magnetics IDS
data.

"""

from PyQt5.QtGui import QIcon, QPixmap
from PyQt5.QtDesigner import QPyDesignerCustomWidgetPlugin

from exampleWidget import exampleWidget

class exampleplugin(QPyDesignerCustomWidgetPlugin):
    """Plugin for exampleWidget functionality.
    """
    def __init__(self, parent=None):
        super(exampleplugin, self).__init__(parent)

    def createWidget(self, parent):
        return exampleWidget(parent=parent, ids=None)

    def name(self):
        return "exampleWidget"

    def group(self):
```

(continues on next page)

(continued from previous page)

```
        return "IMASViz"

    def icon(self):
        return QIcon(_logo_pixmap)

    def toolTip(self):
        return "Plot magnetics IDS data."

    def whatsThis(self):
        return ""

    def isContainer(self):
        return False

    def domXml(self):
        return '<widget class="exampleWidget" name="exampleWidget">\n</widget>'

    def includeFile(self):
        return "exampleWidget"

# Define the image used for the icon.
# Note: This is IMASviz default pixmap.
_logo_16x16_xpm = [
    "16 16 3 1 ",
    "  c black",
    ". c #0000E3",
    "X c None",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX",
    " XX X  XX  XX  X",
    " XX    XX  XX XX",
    " XX  X X   XXX  ",
    " XX XX X XX X  X",
    "XXXXXXXXXXXXXXXX",
    "................",
    "XXXXXXXXXXXXXXXX",
    "XXX XX X X   XXX",
    "XXX X XX XX XXXX",
    "XXXX  XX X XXXXX",
    "XXXX  XX X   XXX",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX"]

_logo_pixmap = QPixmap(_logo_16x16_xpm)
```

Below are listed lines of the Qt plugin code, which must be modified for any new widget, in order to properly refer to the widget source code - in this case **exampleWidget** (exampleWidget.py).

1. Import statement:

```
from exampleWidget import exampleWidget
```

2. Class label:

```
class exampleplugin(QPyDesignerCustomWidgetPlugin):
```

3. Class constructor:

```
    def __init__(self, parent=None):
        super(exampleplugin, self).__init__(parent)
```

4. Returning custom widget object on *createWidget*.

Note: If widget constructor requires arguments they must be included here! In this case *parent* and *ids*.

```
    def createWidget(self, parent):
        return exampleWidget(parent=parent, ids=None)
```

5. Name:

```
    def name(self):
        return "exampleWidget"
```

6. Group:

```
    def group(self):
        return "IMASViz"
```

7. Tool tip:

```
    def toolTip(self):
        return "Plot magnetics IDS data."
```

8. XML attribute definition:

```
    def domXml(self):
        return '<widget class="exampleWidget" name="exampleWidget">\n</widget>'
```

9. Include file:

```
    def includeFile(self):
        return "exampleWidget"
```

10. Icon - pixmap (optional):

```
# Define the image used for the icon.
# Note: This is IMASviz default pixmap.
_logo_16x16_xpm = [
    "16 16 3 1 ",
    "  c black",
    ". c #0000E3",
    "X c None",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX",
    " XX X  XX  XX  X",
    " XX    XX  XX XX",
    " XX  X X   XXX  ",
    " XX XX X XX X  X",
    "XXXXXXXXXXXXXXXX",
    "................",
    "XXXXXXXXXXXXXXXX",
    "XXX XX X X   XXX",
```

(continues on next page)

(continued from previous page)

```
    "XXX X XX XX XXXX",
    "XXXX  XX X XXXXX",
    "XXXX  XX X   XXX",
    "XXXXXXXXXXXXXXXX",
    "XXXXXXXXXXXXXXXX"]

_logo_pixmap = QPixmap(_logo_16x16_xpm)
```

With the source and plugin code (.py files) completed they are ready to be used in Qt designer.

To achieve that, first the location of the necessary files must be provided to the Qt designer. This is done by adding a path to the `$PYQTDESIGNERPATH` system environment variable.

```
# Bash shell
export PYQTDESIGNERPATH=/path/to/source/files:${PYQTDESIGNERPATH}
# C-shell
setenv PYQTDESIGNERPATH /path/to/source/files:${PYQTDESIGNERPATH}
```

in this case

```
# Bash shell
export PYQTDESIGNERPATH=$VIZ_HOME/imasviz/VizPlugins/viz_example:${PYQTDESIGNERPATH}
# C-shell
setenv PYQTDESIGNERPATH $VIZ_HOME/imasviz/VizPlugins/viz_example:${PYQTDESIGNERPATH}
```

With this step completed the PyQt5 widget, now Qt Designer plugin, is ready to be used within **Qt Designer**.

3.1.3 Creating a custom application/plugin with Qt Designer

In this subsection, the process of creating a custom plugin/application GUI is presented. In **Qt Designer**, the GUI design and layout can be done conveniently with mouse drag-and-drop, popup-menu configurations and more.

Note: A good video presentation how to use Qt Designer is available in *Developing a custom user interface (UI) plugins with Qt designer*.

The figure below presents the final look at the end of the plugin GUI design procedure.

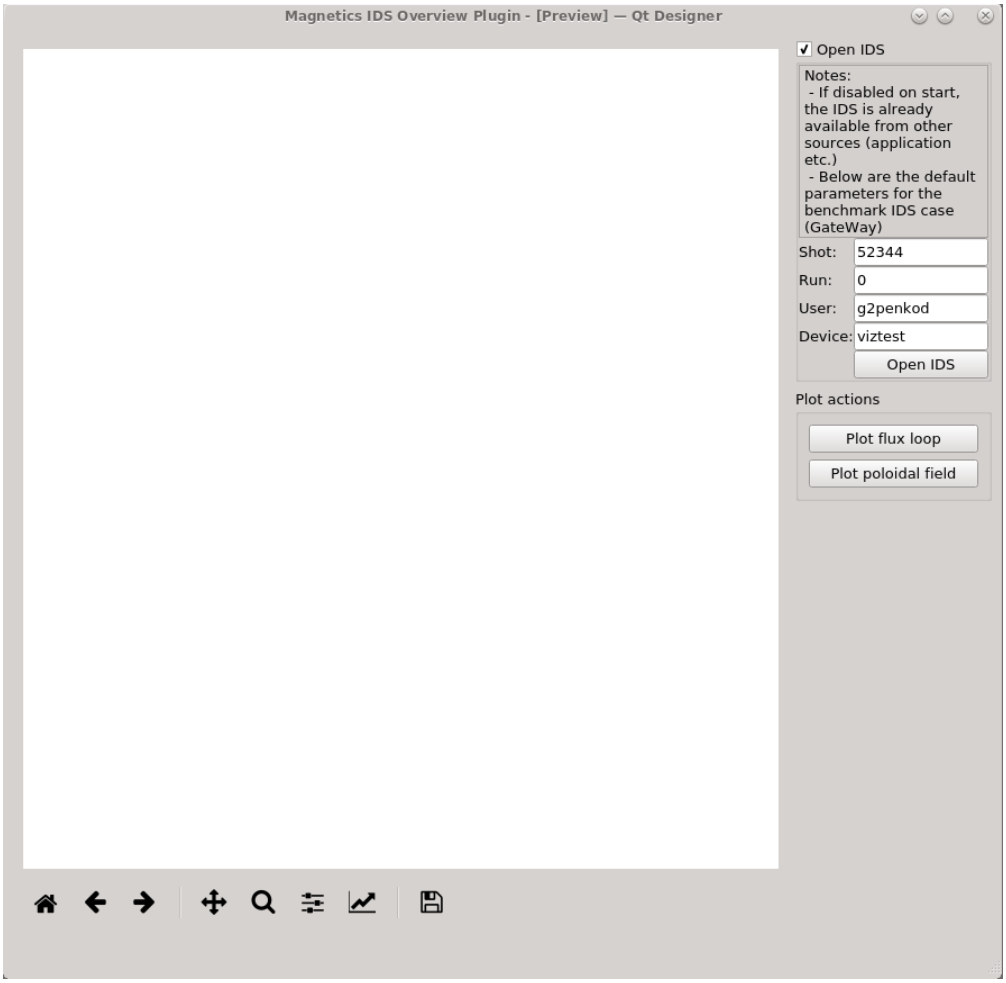


Fig. 3: Final design of the example plugin, indented for plotting all slices of **flux loop** and **poloidal field** data found in **magnetics IDS**.

Qt designer is (usually) run with

```
designer-qt5
```

Warning: Qt version of used PyQt5 (compiled with Qt) and Qt designer **must match**! Qt designer with Qt version X will not be able to find a plugin which source (widget code) was written using PyQt5 compiled with Qt version Y!

A startup window will appear, as shown in the figure below.

3.1.3.1 GUI design procedure

1. First, a new **Main Window** must be created. This is done by selecting The **Main Window** option from the list of *templates/forms* and clicking the *Create* button, as shown in the figure below.

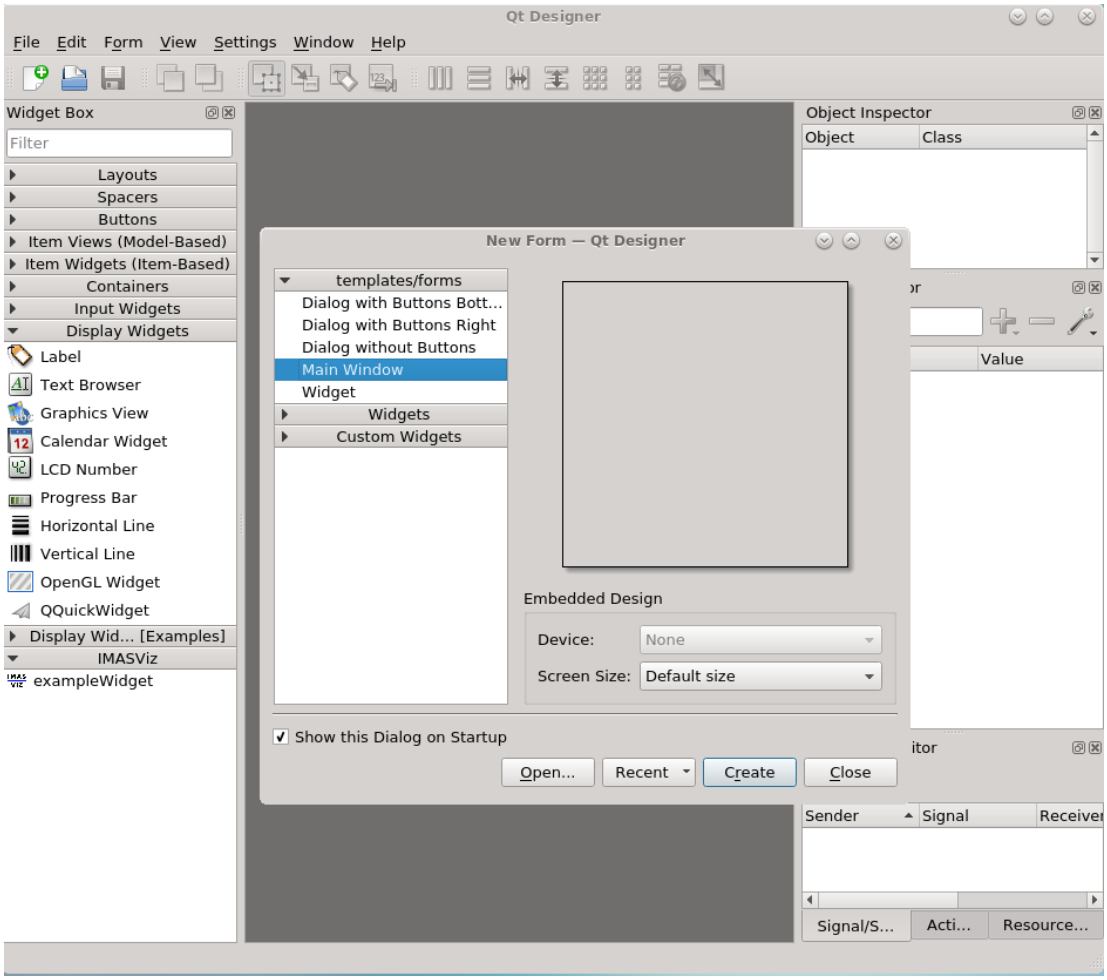


Fig. 4: Qt designer startup window.

After this is done, on the far left side of the window lays a **Widget Box** which displays a collection of all available widgets. On the bottom of the list, a group **IMASViz** containing widget labeled **exampleWidget** can be found. This is the custom widget which was developed through the first half of this manual section. The group **IMASViz** was defined in the **plugin.py** file (**def group**).

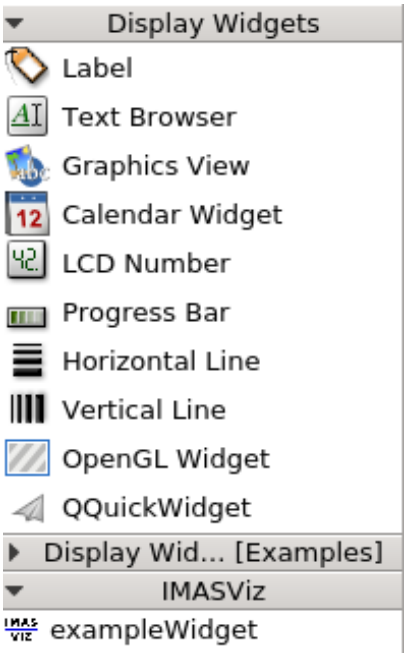


Fig. 5: Custom widget (`exampleWidget`) in Qt designer.

2. Next, drag and drop **exampleWidget** to *MainWindow*. The result should be similar as in the figure below.

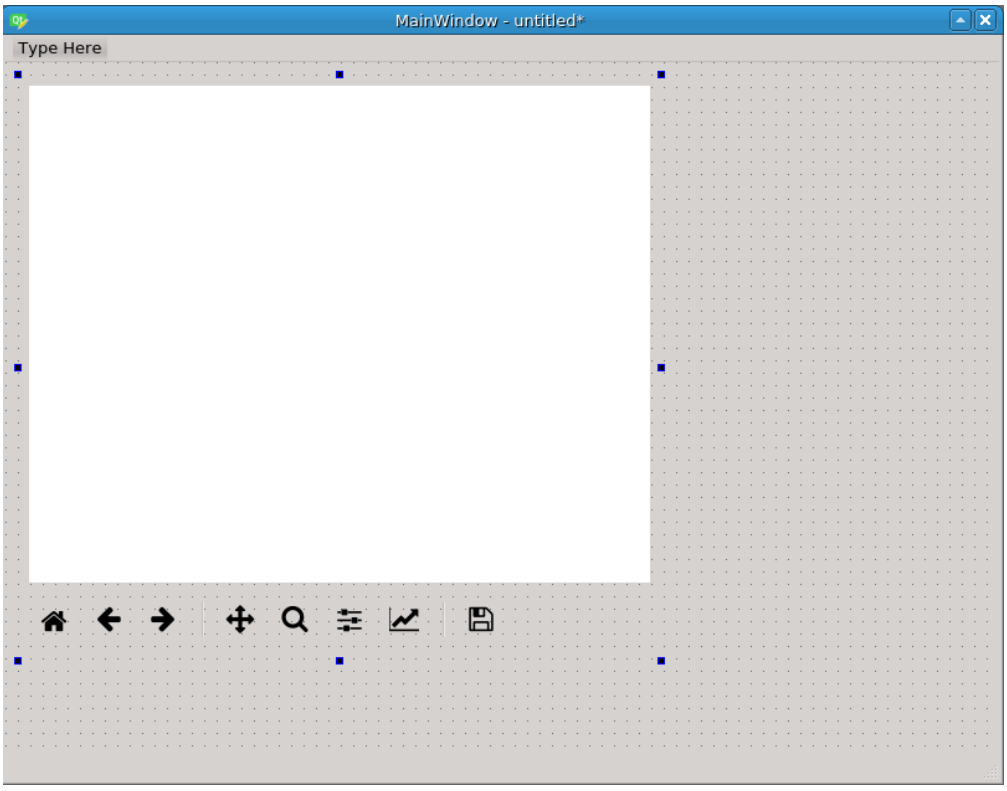


Fig. 6: **exampleWidget** within *MainWindow*.

3. Next, drag and drop 2x *Group Box* and 2x *vertical spacer*.

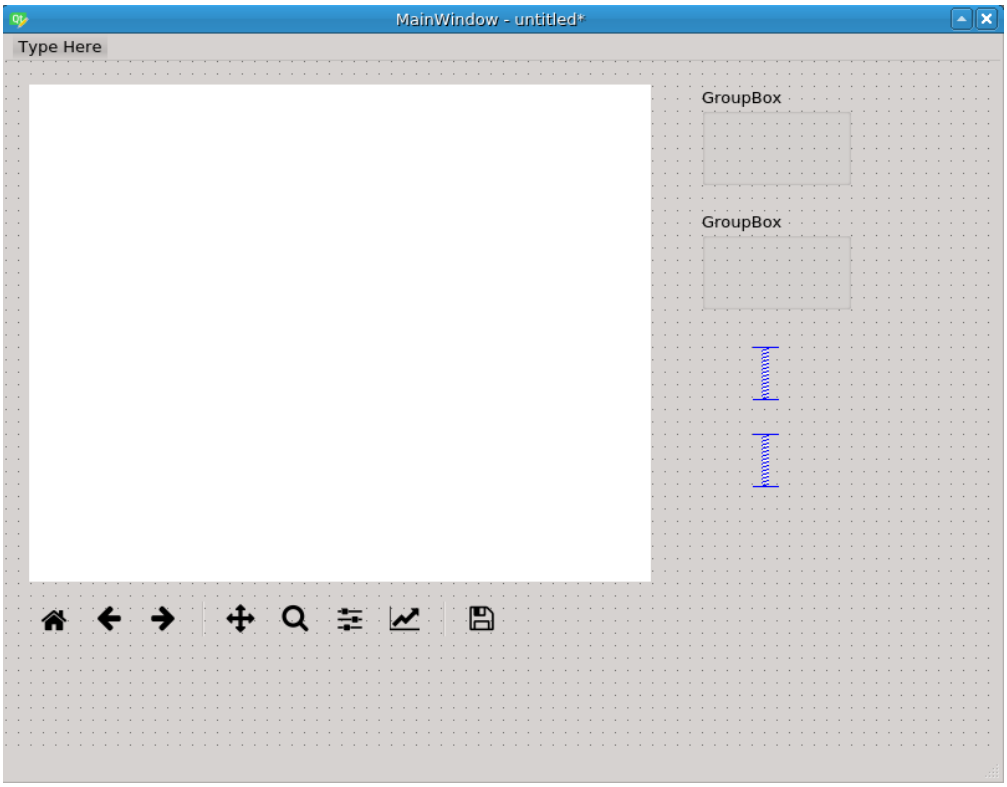


Fig. 7: Added 2x *Group Box* and 2x *vertical spacer* to *MainWindow*.

4. In top *Group Box*:

   4.1. Add 5x *Label*, 4x *LineEdit* and 1x *Push Button* widgets.

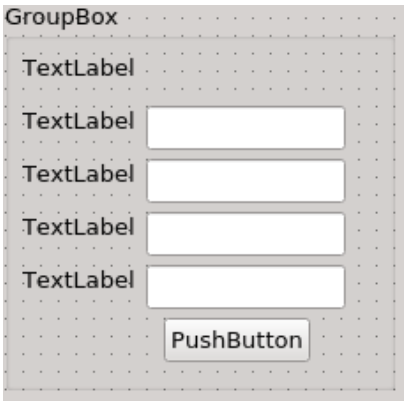


Fig. 8: Rough *Group Box* design and layout.

   4.2. Right click within the box and select *Layout -> Lay Out in a Grid*.

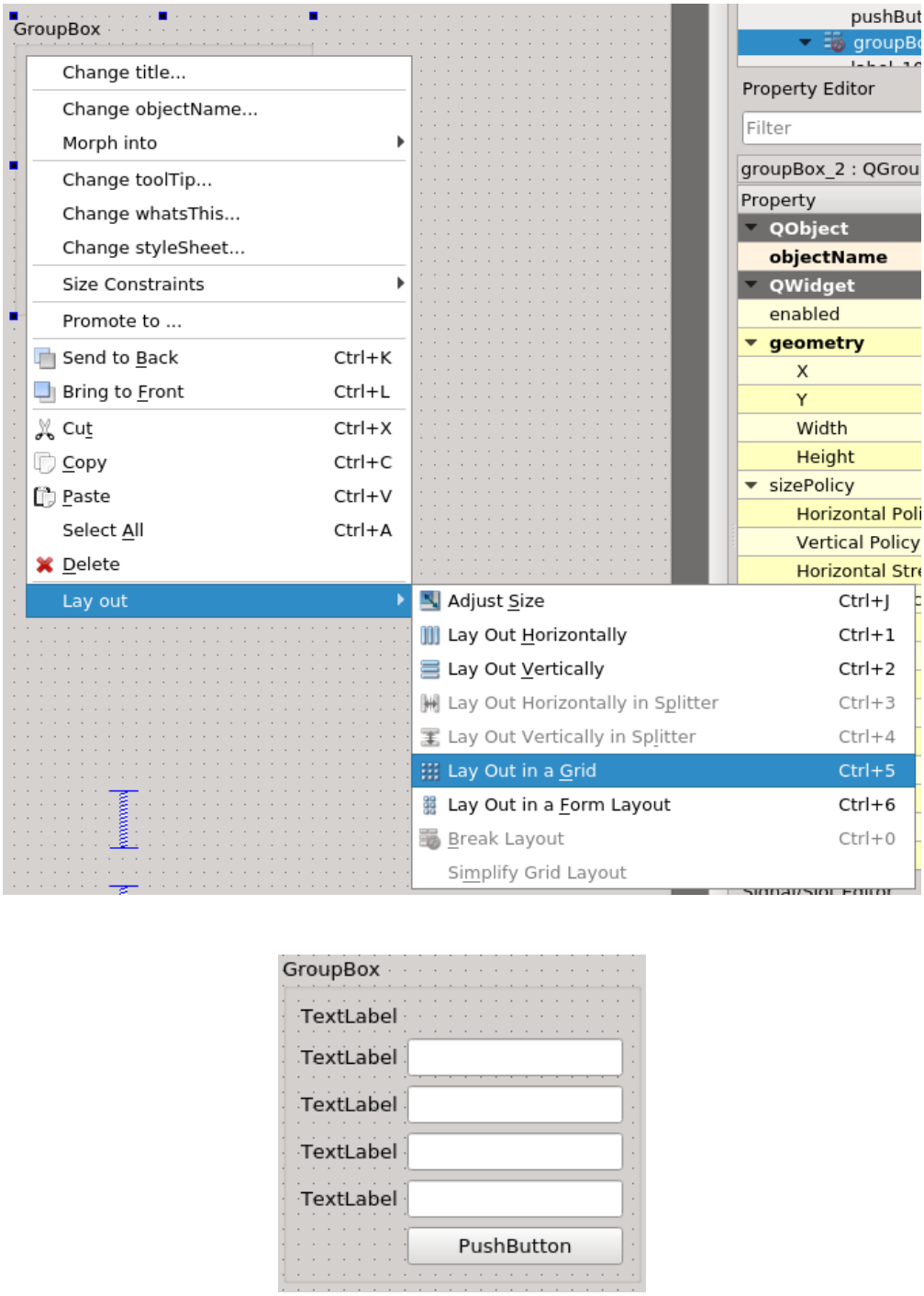


Fig. 9: Grid layout in the top *Group Box*.

4.3. Set suitable texts to *groupbox*, *Label* and *Push Button* widgets. Set default values to *LineEdit* widgets.

Note: Top *Label* widget contains the next text:

Notes: \n - If disabled on start, the IDS is already available from other sources (application etc.)

\n - Below are the default parameters for the benchmark IDS case (GateWay)

\n are required to achieve line breaks.

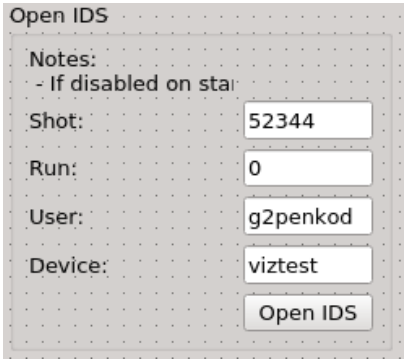


Fig. 10: Top *Group Box* with set labels and default values.

4.4. Select *Group Box* and change its next properties in the *Property Editor* found on the right side of the Qt Designer application:

- QGroupBox -> checkable = True (check)

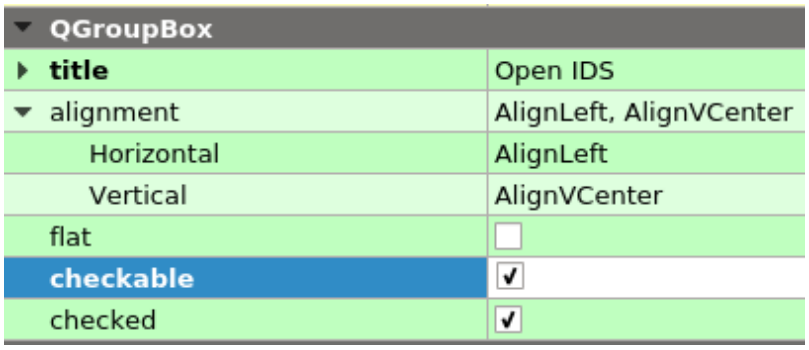


Fig. 11: *Group Box* property `checkable` found in the *Property Editor*.

- QWidget -> sizePolicy -> Horizontal policy = Minimum
- QWidget -> maximumSize -> Width = 175
- Layout -> layoutLeftMargin = 0
- Layout -> layoutTopMargin = 0
- Layout -> layoutRightMargin = 0
- Layout -> layoutBottomMargin = 0
- Layout -> layoutHorizontalSpacing = 0
- Layout -> layoutVerticalSpacing = 0

4.5. Select the top *Label* widget and change its next properties in the *Property Editor*:

- QFrame -> frameShape = StyledPanel
- QLabel -> wordwrap = True (checked)

Note: Manually (with mouse cursor) resize the *Group Box* and the *Label* to see the whole text of the top label.

4.6 Set next properties to all *LineEdit* widgets:

- QWidget -> sizePolicy -> Horizontal policy = Minimum
- QWidget -> sizePolicy -> Vertical policy = Fixed

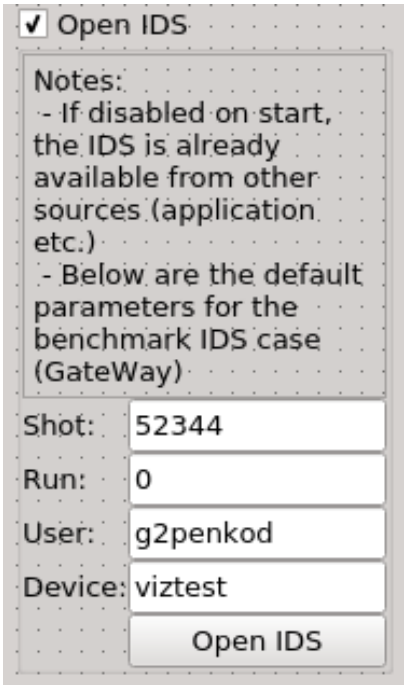


Fig. 12: Finished top *Group Box*.

5. In bottom *Group Box*:

5.1. Add 2x *Push Button* widget.

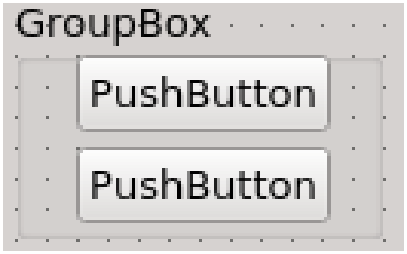


Fig. 13: Bottom *Group Box* template.

5.2. Label the *Group Box* and *Push Button* widgets.

5.3. Right click within the box and select *Layout -> Lay Out in a Grid*.

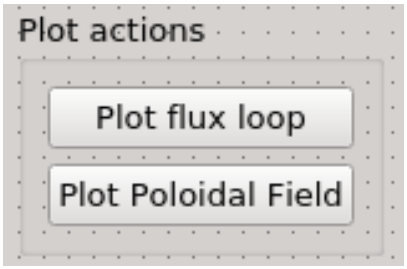


Fig. 14: Finished bottom *Group Box*.

6. Right click within the *MainWindow* and select *Layout* -> *Lay Out in a Grid*.

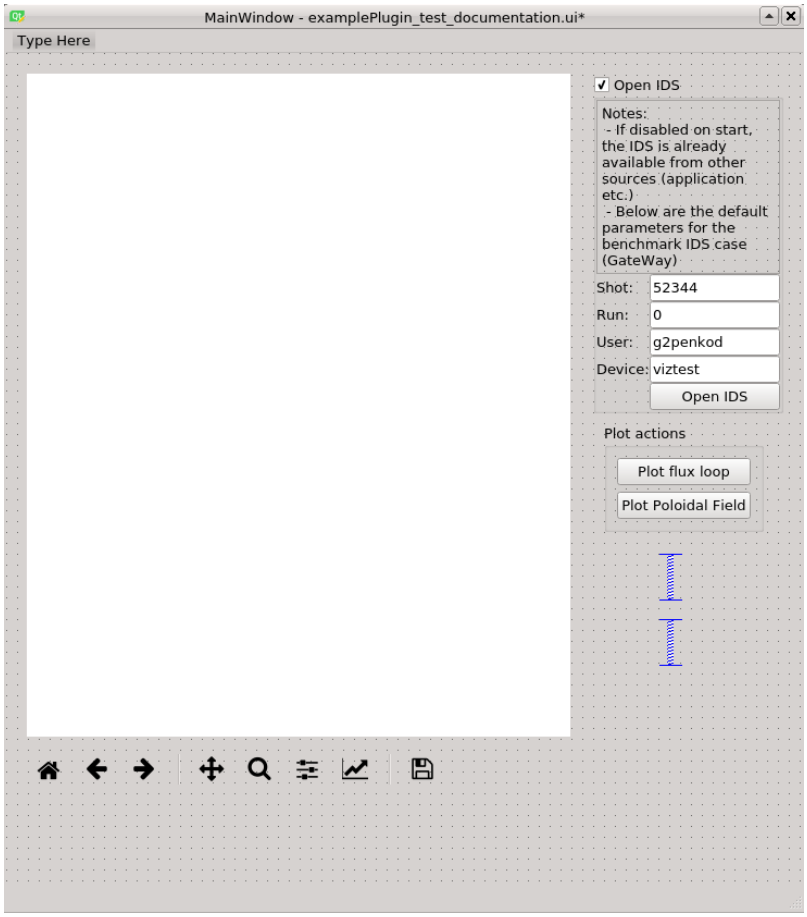


Fig. 15: *MainWindow* before setting grid layout.

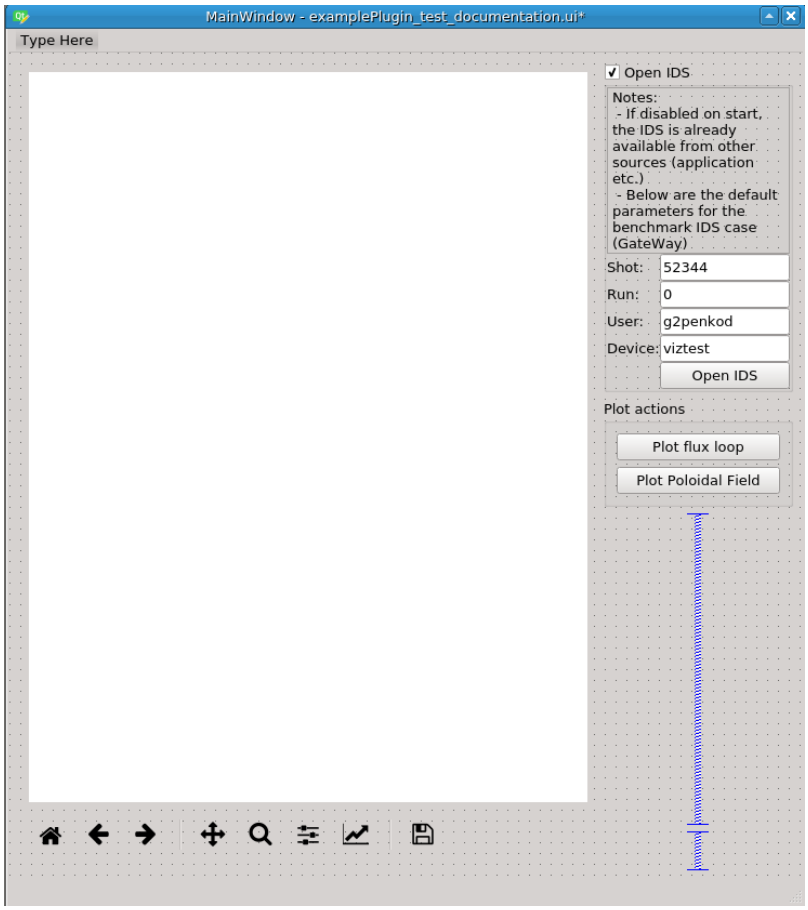


Fig. 16: *MainWindow* after setting grid layout.

7. Change *MainWindow* properties:
   - QWidget -> windowTitle = Magnetics IDS Overview Plugin
8. Change **exampleWidget** properties:
   - QObject -> objectName = mainPluginWidget

> **Warning:** This property definition is crucial in the later sections in this HowTo manual when linking the plugin in IMASViz.

   - QWidget -> sizePolicy -> HorizontalPolicy = Expanding
   - QWidget -> sizePolicy -> VerticalPolicy = Expanding

3.1.3.2 Edit signals/slots

By editing **signals/slots** the wanted actions such as plot execution etc. are added to the widgets.

To edit **signals/slots**, in menubar, navigate to **Edit** -> **Edit Signals/Slots**

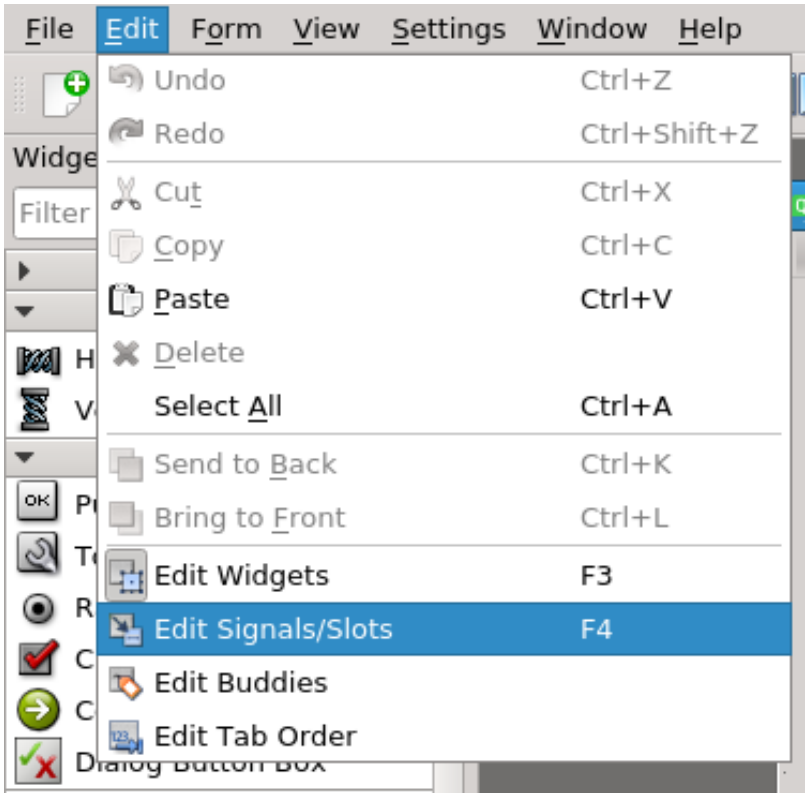


Fig. 17: **Edit Signals/Slots** in menubar.

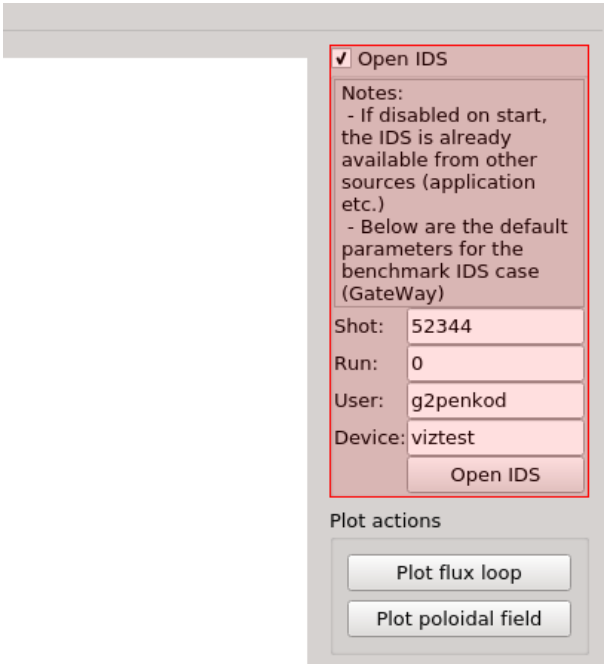
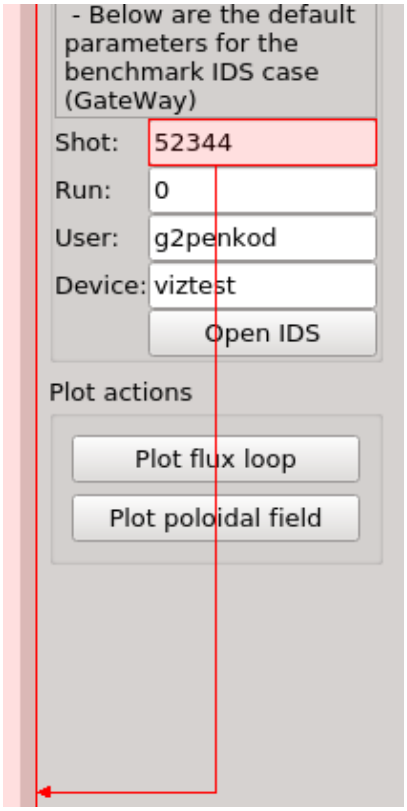


Fig. 18: While in **Edit Signals/Slots mode**, hovering with the mouse over widgets marks them with slight red color.

Link **Line Edit** widgets (located in the top *Group Box*) signals to **exampleWidget** slots. This is done by clicking on

one of the **Line Edit** widgets (in this case that one which holds the **Shot** value), holding and dragging the shown arrow to **exampleWidget**, as shown in the figure below.



Next, the **Configure connection editor** will be shown.

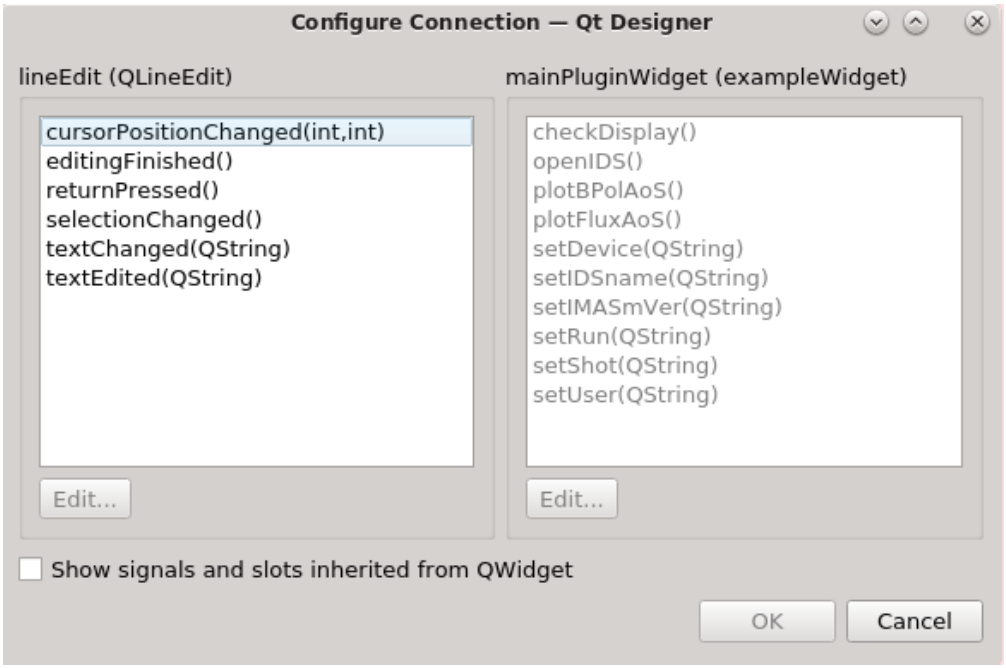


Fig. 19: **Configure connection editor**, displaying list of available **Line Edit** (left) **signals** and available **exampleWidget** slots (right).

The slots are in this case actually functions/routines, which were defined in the **exampleWidget** source code (done in section *Widget class*).

In this case, when editing the text in **Line Edit** and applying the changes (pressing `enter` key etc.) the changed value must be passed to the **exampleWidget**. This is done by selecting the suitable signal and slot from the lists: **textEdited** and **setShot**, and pressing the *OK* button.

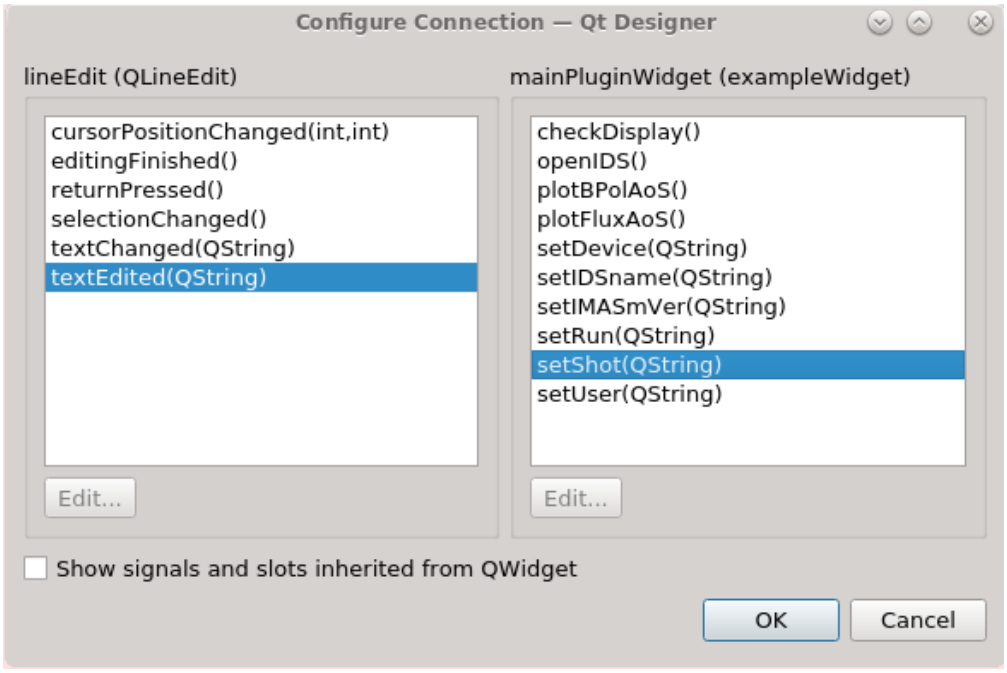


Fig. 20: Selection of **Line Edit** signal **textEdited** and **exampleWidget** slot **setShot**

After that, while in the **Edit Signals/Slots**, the start and the end point of the red arrow will indicate which signal and slot are linked.

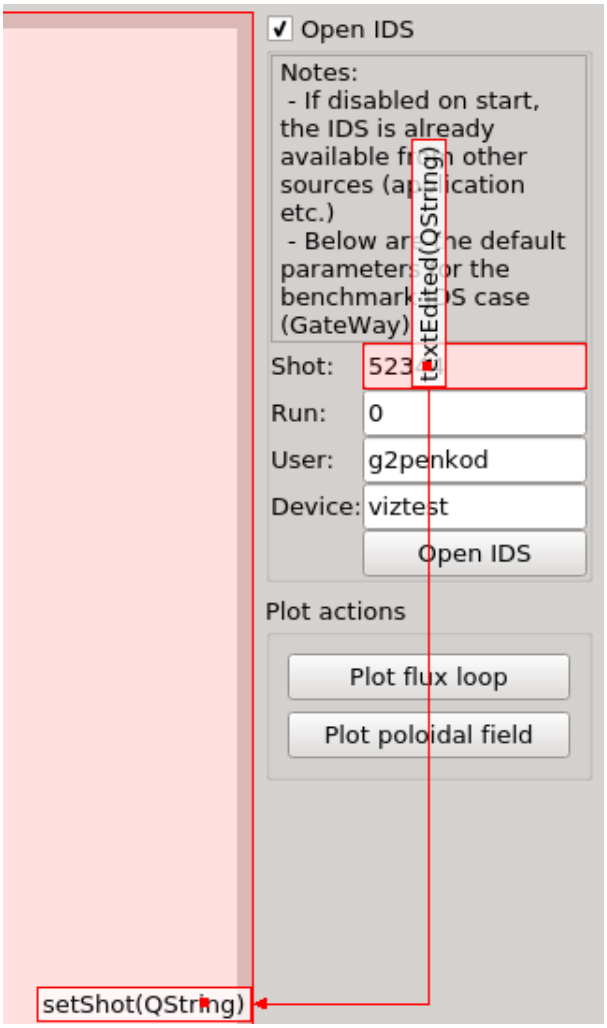


Fig. 21: Red arrow indicating the **sender**, **sender signal**, **receiver** and **slot**.

The whole list of created slot/signal links is shown on the bottom right corner of the Qt Designer application.

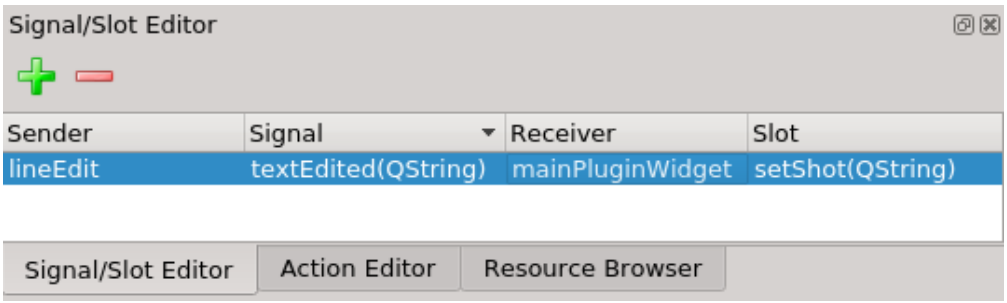


Fig. 22: A list displaying a list of all created slot/signal lists, listing the **sender**, **sender signal**, **receiver** and **slot** for each link.

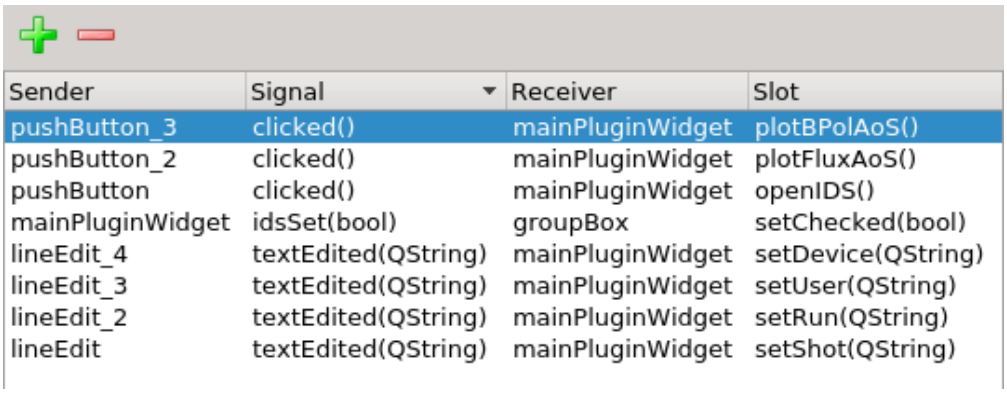
The same as the first link, few more links are required. Below is a detailed table listing all necessary links.

For easier interpretation, the **sender** and **receiver** in the table below are marked with their **text label** instead of their

object name (used in the Qt Designer list of signal/slots).

SENDER			RECEIVER	
Type	Label/Value	Signal	Type	Signal
Line Edit	52344	textEdited(QString)	exampleWidget	setShot(QString)
Line Edit	0	textEdited(QString)	exampleWidget	setRun(QString)
Line Edit	g2penkod	textEdited(QString)	exampleWidget	setUser(QString)
Line Edit	viztest	textEdited(QString)	exampleWidget	setDevice(QString)
Push Button	Open IDS	clicked()	exampleWidget	openIDS()
Push Button	Plot flux loop	clicked()	exampleWidget	openIDS()
Push Button	Plot poloidal field	clicked()	exampleWidget	openIDS()
exampleWidget	Open IDS	idsSet()	Group Box (top)	SetChecked(bool)

The final list of necessary signal/slot links in Qt Designer for this case is shown in the figure below.

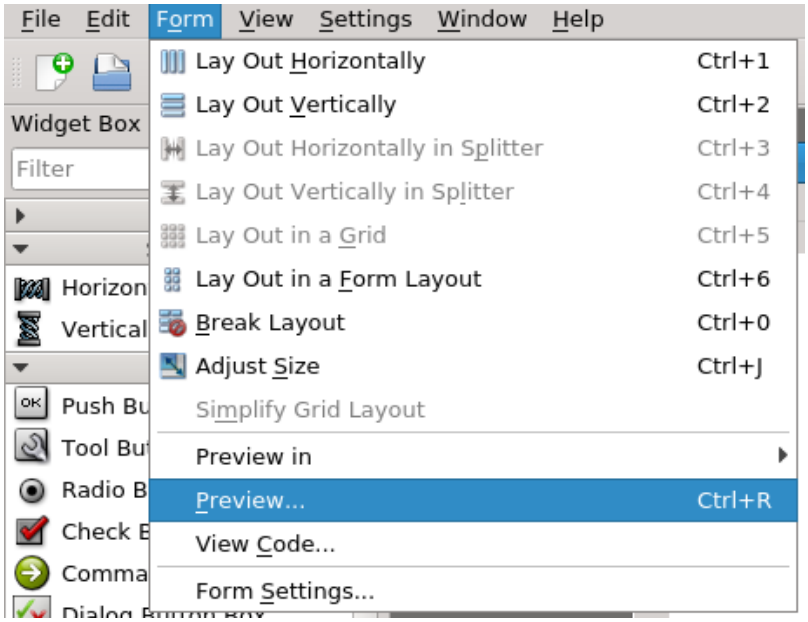


Sender	Signal	Receiver	Slot
pushButton_3	clicked()	mainPluginWidget	plotBPolAoS()
pushButton_2	clicked()	mainPluginWidget	plotFluxAoS()
pushButton	clicked()	mainPluginWidget	openIDS()
mainPluginWidget	idsSet(bool)	groupBox	setChecked(bool)
lineEdit_4	textEdited(QString)	mainPluginWidget	setDevice(QString)
lineEdit_3	textEdited(QString)	mainPluginWidget	setUser(QString)
lineEdit_2	textEdited(QString)	mainPluginWidget	setRun(QString)
lineEdit	textEdited(QString)	mainPluginWidget	setShot(QString)

At this point, the plugin is completed.

3.1.3.3 Qt Designer Preview

The constructed plugin can be tested with the Qt Designer **Preview** option, found in the **Form** menu.



By pressing first the *Open IDS* button, waiting for a moment until the IDS data gets read, and then pressing the *Plot flux loop* button, the plot panel is populated as shown in the figure below.

Warning: This specified case is done for the GateWay HPC. Make sure, that the corresponding IDS exists and that it contains the right data!

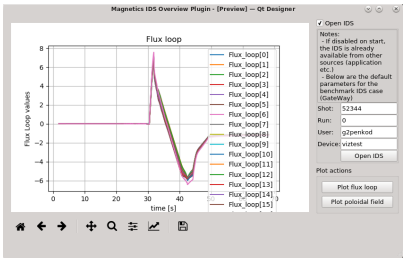


Fig. 23: Plotting all **Flux loop** plots from the **magnetics IDS** with parameters (on GateWay HPC!) **Shot:** 52344, **Run:** 0, **user**: g2penkod, **device**: viztest.

3.1.3.4 Saving the Qt Designer form

The created Qt Designer GUI form (**.ui** extension) can be saved by navigating from menubar to **File -> Save**. The name can be set customly, in this case it is saved as **examplePlugin.ui** (do not confuse it with **examplePlugin.py**).

Warning: **IMPORTANT: the .ui file must be saved in the same directory as the source files**, in this case **exampleWidget.py** and **examplePlugin.py**.

3.1.3.5 Running the plugin .ui

The **.ui** plugin can be run using Python3 shell. Open Python3 shell and type the next commands:

```
from PyQt5.QtWidgets import QApplication
from PyQt5 import uic
app = QApplication([])
uiObj = uic.loadUi('SOLPSplugin.ui')
uiObj.show()
```

A script example `run_plugin_ui_standalone.py` is available in the plugin directory. It is run with the following command:

```
python3 run_plugin_ui_standalone.py
```

3.1.4 Adding plugin to IMASViz

Warning: Before proceeding, make sure that you made the step 8. in *Creating a custom application/plugin with Qt Designer*. Setting the **objectName** of the custom widget to **mainPluginWidget** is mandatory!

The main idea for integration of plugin in IMASViz is to simplify the plugin usage and to add further functionalities to IMASViz. By running the plugin from IMASViz the IMASViz created **IDS object** is passed to the plugin, thus opening and setting the IDS is not necessary (required when running the plugin as a standalone application as shown in *Running the plugin .ui*).

To run the plugin from IMASViz it must be first added (registered) in IMASViz `$VIZ_HOME/imasviz/VizPlugins/VizPlugins.py` source file. This is done through the next few steps:

1. Add plugin to a list of registered plugins

The **RegisteredPlugins** dictionary contains major plugin properties. The relevant properties for **examplePlugin** are highlighted in the code block below, where:

- **example_UiPlugin** is a dictionary key (dictionary within **RegisteredPlugins** dictionary).

Warning: Mandatory: in case the plugin is created with the help of **Qt Designer** and **.ui** file is created, the key must contain suffix **_UiPlugin** in order for IMASViz to recognize and use it correctly!

- **UiFile** dictionary key, holding full **.ui** filename.
- **dir** dictionary key, holding the path to the dictionary where the **.ui** filename (and other plugin sources) are located.
- **targetIDSroot** dictionary key, holding target **IDS** label.
- **targetOccurrence** dictionary key, holding target **IDS occurrence** integer.

```
RegisteredPlugins = {'equilibriumcharts':'viz_equi.equilibriumcharts',
                     'SOLPS_UiPlugin': {
                         'UiFile': 'SOLPSplugin.ui',
                         'dir': os.environ['VIZ_HOME'] +
                                '/imasviz/VizPlugins/viz_solps/',
                         'targetIDSroot' : 'edge_profiles',
                         'targetOccurrence' : 0},
                     'example_UiPlugin': {
                         'UiFile': 'examplePlugin.ui',
                         'dir': os.environ['VIZ_HOME'] +
                                '/imasviz/VizPlugins/viz_example/',
```

(continues on next page)

(continued from previous page)

```
                            'targetIDSroot': 'magnetics',
                            'targetOccurrence': 0}
                        }
```

2. Add plugin configuration

Each plugin can have its own specific configuration. In the case of the **examplePlugin** there are no configurations required. Still, an empty configuration must be provided, as highlighted in the code block below.

```
RegisteredPluginsConfiguration = {'equilibriumcharts':[{
                                        'time_i': 31.880, \
                                        'time_e': 32.020, \
                                        'delta_t': 0.02, \
                                        'shot': 50642, \
                                        'run': 0, \
                                        'machine': 'west_equinox', \
                                        'user': 'imas_private'}],
                                    'SOLPS_UiPlugin':[{}],
                                    'example_UiPlugin':[{}]
                                }
```

3. Add necessary entries

The entries are mainly related to plugin identification and presentation in IMASViz in terms of label in pop-up menus. The required entries are highlighted in the code blocks below.

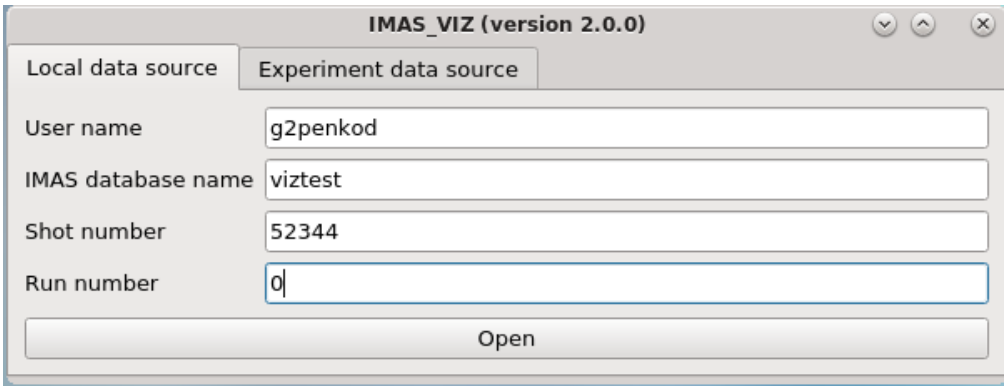
```
EntriesPerSubject = {'equilibriumcharts': {'equilibrium_overview': [0],
                                           'overview': [0]},
                     'ToFuPlugin':        {'interferometer_overview': [0, 1],
                                           'bolometer_overview': [2, 3],
                                           'soft_x_rays_overview': [4, 5]},
                     'SOLPS_UiPlugin':    {'edge_profiles_overview':[0],
                                           'overview':[0]},
                     'example_UiPlugin':  {'magnetics_overview': [0],
                                           'overview': [0]}
                     }
```

```
AllEntries = {'equilibriumcharts': [(0, 'Equilibrium overview...')],
              'ToFuPlugin':        [(0, 'tofu - geom...'), (1, 'tofu - data'),
                                    (2, 'tofu - geom...'), (3, 'tofu - data'),
                                    (4, 'tofu - geom...'), (5, 'tofu - data')],
              'SOLPS_UiPlugin':    [(0, 'SOLPS overview...')],
              'example_UiPlugin':  [(0, 'Magnetics overview...')]
              }
```

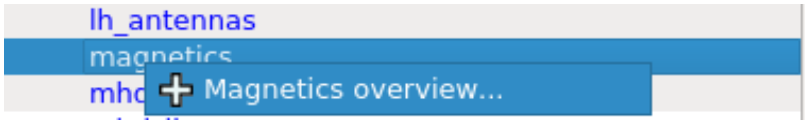
Now everything is ready to run the plugin from IMASViz.

3.1.5 Running the custom plugin in IMASViz

When running the IMASViz, for the means of this manual, open the IDS with the same case parameters as defined in *Edit signals/slots*.



In the *tree window*, navigate to *magnetics*. While holding **shift key** right click on the *magnetics* label and in the popup menu select the *Magnetics overview...* option.



On selection confirm, the **examplePlugin**, now referred to as **Magnetics IDS Overview Plugin**, the plugin window is shown.

Fig. 24: **Magnetics IDS Overview Plugin** on startup when run from within **IMASViz**.

It can be observed, that the top *Group Box* is disabled. This is due to our code and signal/slots, done in the plugin development phase in previous sections. This way if IDS object (now provided by IMASViz) is already provided on plugin startup the IDS set/open/read procedures are not required, but if needed are still functional and can be enabled with checking the checkbox. This way the plugin can be conveniently used as standalone or within other applications.

By pressing either *Plot flux loop* or *Plot poloidal field* buttons the corresponding data, specified in the plugin code development phase, are plotted.

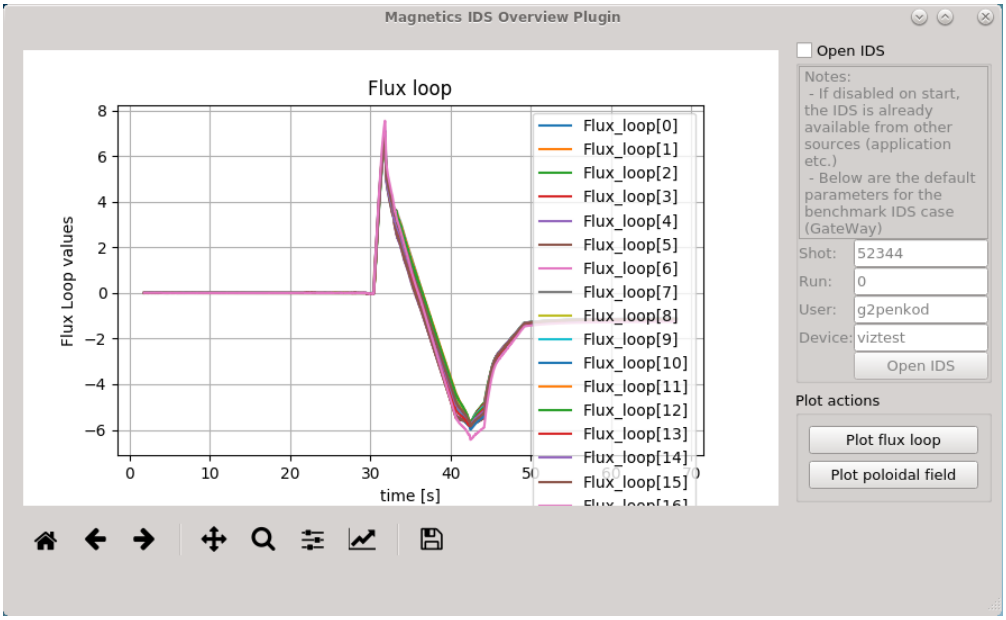


Fig. 25: **Magnetics IDS Overview Plugin** with plotted all **Flux loop** data from the **magnetics IDS** (parameters (on GateWay HPC!) **Shot:** 52344, **Run:** 0, **user**: g2penkod, **device**: viztest.