DataLab

Release 1.0.1

Nov 18, 2025

CONTENTS

1 Getting started **3**
- 1.1 Installation . . . 3
- 1.2 Use cases, main features and key strengths . . . 9
- 1.3 Key features . . . 12
- 1.4 Tutorials . . . 14

2 Features **87**
- 2.1 Overview & Common features . . . 87
- 2.2 Signal processing . . . 102
- 2.3 Image processing . . . 141
- 2.4 Validation . . . 185
- 2.5 Advanced features . . . 194

3 Contributing **309**
- 3.1 Share your ideas and experiences . . . 309
- 3.2 Share your scientific/technical knowledge . . . 309
- 3.3 Contribute to new features . . . 310
- 3.4 Develop new features . . . 310

4 Release notes **327**
- 4.1 DataLab Version 1.0.1 . . . 327
- 4.2 DataLab Version 0.20.0 . . . 335
- 4.3 DataLab Version 0.19.2 . . . 337
- 4.4 DataLab Version 0.19.1 . . . 337
- 4.5 DataLab Version 0.19.0 . . . 338
- 4.6 DataLab Version 0.18.2 . . . 340
- 4.7 DataLab Version 0.18.1 . . . 341
- 4.8 DataLab Version 0.18.0 . . . 342
- 4.9 DataLab Version 0.17.1 . . . 343
- 4.10 DataLab Version 0.17.0 . . . 343
- 4.11 DataLab Version 0.16.4 . . . 345
- 4.12 DataLab Version 0.16.3 . . . 346
- 4.13 DataLab Version 0.16.2 . . . 346
- 4.14 DataLab Version 0.16.1 . . . 347
- 4.15 DataLab Version 0.16.0 . . . 348
- 4.16 DataLab Version 0.15.1 . . . 350
- 4.17 DataLab Version 0.15.0 . . . 351
- 4.18 DataLab Version 0.14.2 . . . 352
- 4.19 DataLab Version 0.14.1 . . . 352
- 4.20 DataLab Version 0.14.0 . . . 353

4.21 DataLab Version 0.12.0 354
4.22 DataLab Version 0.11.0 356
4.23 DataLab Version 0.10.1 359
4.24 DataLab Version 0.9.2 362
4.25 DataLab Version 0.9.1 362
4.26 DataLab Version 0.9.0 363

Python Module Index **367**

DataLab is an **open-source platform for signal and image processing and visualization** for research, education and industry. Leveraging the richness of the scientific Python ecosystem[1], DataLab is the ideal complement to your data analysis workflows as it can be extended with your Python code through *Plugins* or directly from *your IDE* or *your Jupyter notebooks*. Go to *Installation* to get started!

To immediately see DataLab in action, you have two options:

- Read or view our *Tutorials*,
- Try DataLab online, without installation, using our Binder environment.

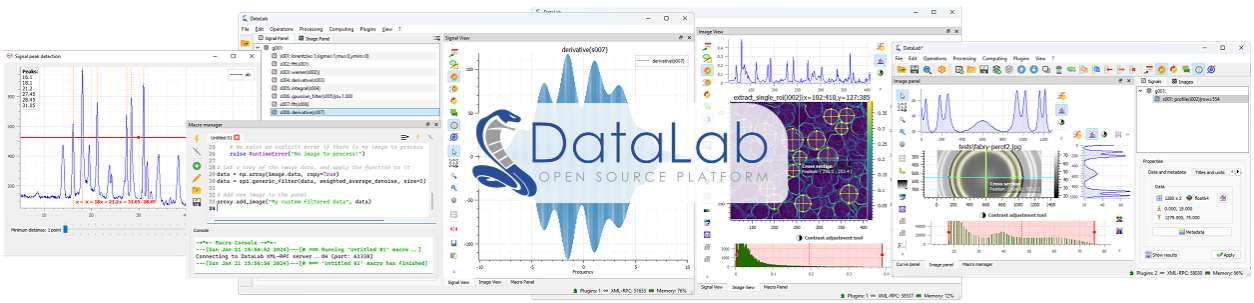


Fig. 1: Signal and image visualization in DataLab

DataLab has been funded, chronologically, by the following stakeholders:

CEA, the French Alternative Energies and Atomic Energy Commission, is the major investor in DataLab, and is the main contributor to the project.
CODRA, a software engineering and editor firm, has supported DataLab open-source journey since its inception (see here).
NLnet Foundation, as part of the NGI0 Commons Fund, backed by the European Commission, has funded the redesign of DataLab's core architecture.

Powered by PlotPyStack

Fig. 2: DataLab is powered by PlotPyStack, the scientific Python-Qt visualization and graphical user interface stack.

Sigima

Fig. 3: DataLab processing features are based on Sigima, the open-source signal and image processing library (part of the DataLab Platform).

[1] DataLab processing primitives are mainly based on Sigima, NumPy, SciPy, scikit-image, OpenCV and PyWavelets libraries. DataLab visualization capabilities are based on PlotPyStack toolkit, a set of Python libraries for building scientific applications with Qt graphical user interfaces. DataLab advanced and specialized processing features are based on Sigima, the open-source signal and image processing library (part of the DataLab Platform).

CHAPTER

ONE

GETTING STARTED

DataLab is an open platform for signal and image processing, designed to be used by scientists, engineers, and researchers in academia and industry, while offering the reliability of industrial-grade software. It is a versatile software that can be used for a wide range of applications, from simple data analysis to complex signal processing and image analysis tasks.

DataLab integrates seemlessly into your workflow thanks to three main operating modes:

	Stand-alone application, with a graphical user interface that allows you to interact with your data and visualize the results of your analysis in real time.
	Python library, allowing you to integrate DataLab functions (or graphical user interfaces) into your own Python scripts and programs or Jupyter notebooks.
	Remotely controlled from your own software, or from an IDE (e.g., Spyder) or a Jupyter notebook, using the DataLab API.

DataLab leverages the power of Python and its scientific ecosystem, through the use of the following libraries:

- Sigima for signal and image processing (part of the DataLab Platform)
- NumPy for numerical computing (arrays, linear algebra, etc.)
- SciPy for scientific computing (interpolation, special functions, etc.)
- scikit-image and OpenCV for image processing
- PyWavelets for wavelet transform
- PlotPyStack for Qt-based interactive data visualization

1.1 Installation

Warning: Important notice for users upgrading from DataLab v0.20 or earlier:

DataLab v1.0 introduces **breaking changes** that are **not backward compatible** with v0.20.

- **Plugins** developed for v0.20 **must be updated** to work with v1.0
- **API changes** affect custom code integrations

For detailed migration information, see the *migration guide*.

This section provides information on how to install DataLab on your system. Once installed, you can start DataLab by running the `datalab` command in a terminal, or by clicking on the DataLab shortcut in the Start menu (on Windows).

See also:

For more details on how to execute DataLab and its command-line options, see *Command line features*.

1.1.1 How to install

DataLab is available in several forms:

- As a *Conda package*.
- As a Python package, which can be installed using the *Package manager pip*.
- Windows As a stand-alone application, which does not require any Python distribution to be installed. Just run the *All-in-one installer* and you're good to go!
- Windows Within a ready-to-use *Python distribution*, based on WinPython.
- As a precompiled *Wheel package*, which can be installed using `pip`.
- As a *Source package*, which can be installed using `pip` or manually.

See also:

Impatient to try the next version of DataLab? You can also install the latest development version of DataLab from the master branch of the Git repository. See *Development version* for more information.

Conda package

GNU/Linux Windows macOS

To install `datalab` package from the *conda-forge* channel (https://anaconda.org/conda-forge/datalab), run the following command:

```
$ conda install conda-forge::datalab
```

Package manager `pip`

GNU/Linux Windows macOS

DataLab's package `datalab-platform` is available on the Python Package Index (PyPI) on the following URL: https://pypi.python.org/pypi/datalab-platform.

Installing DataLab from PyPI with Qt is as simple as running this command (you may need to use `pip3` instead of `pip` on some systems):

```
$ pip install datalab-platform[qt]
```

Or, if you prefer, you can install DataLab without the Qt library (not recommended):

```
$ pip install datalab-platform
```

Note: If you already have a previous version of DataLab installed, you can upgrade it by running the same command with the `--upgrade` option:

```
$ pip install --upgrade datalab-platform[qt]
```

All-in-one installer

Windows

DataLab is available as a stand-alone application for Windows, which does not require any Python distribution to be installed. Just run the installer and you're good to go!

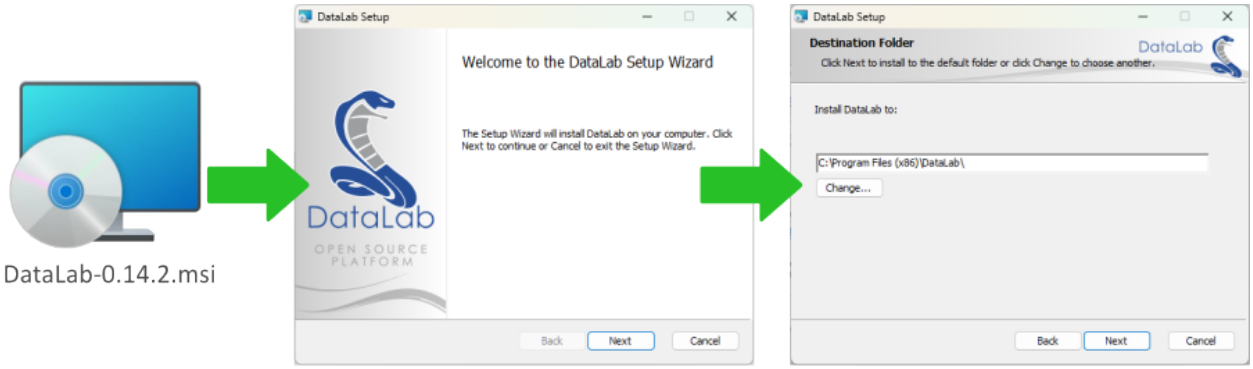


Fig. 1: DataLab all-in-one installer for Windows

The installer package is available in the Releases section. It supports automatic uninstall and upgrade feature (no need to uninstall DataLab before runinng the installer of another version of the application).

> **Warning:** DataLab Windows installer is available for Windows 7 SP1, 8, 10 and 11.
>
> On Windows 7 SP1, before running DataLab (or any other Python 3 application), you must install Microsoft Update *KB2533623* (*Windows6.1-KB2533623-x64.msu*) and also may need to install Microsoft Visual C++ 2015-2022 Redistribuable package.

Python distribution

Windows

DataLab is also available within a ready-to-use Python distribution, based on WinPython. This distribution is called DataLab-WinPython and is available in the DataLab-WinPython Releases section.

Fig. 2: DataLab-WinPython is a ready-to-use Python distribution including the DataLab platform.

The main difference with the all-in-one installer is that you can use the Python distribution for other purposes than running DataLab, and you may also extend it with additional packages. On the downside, it is also *much bigger* than the all-in-one installer because it includes a full Python distribution.

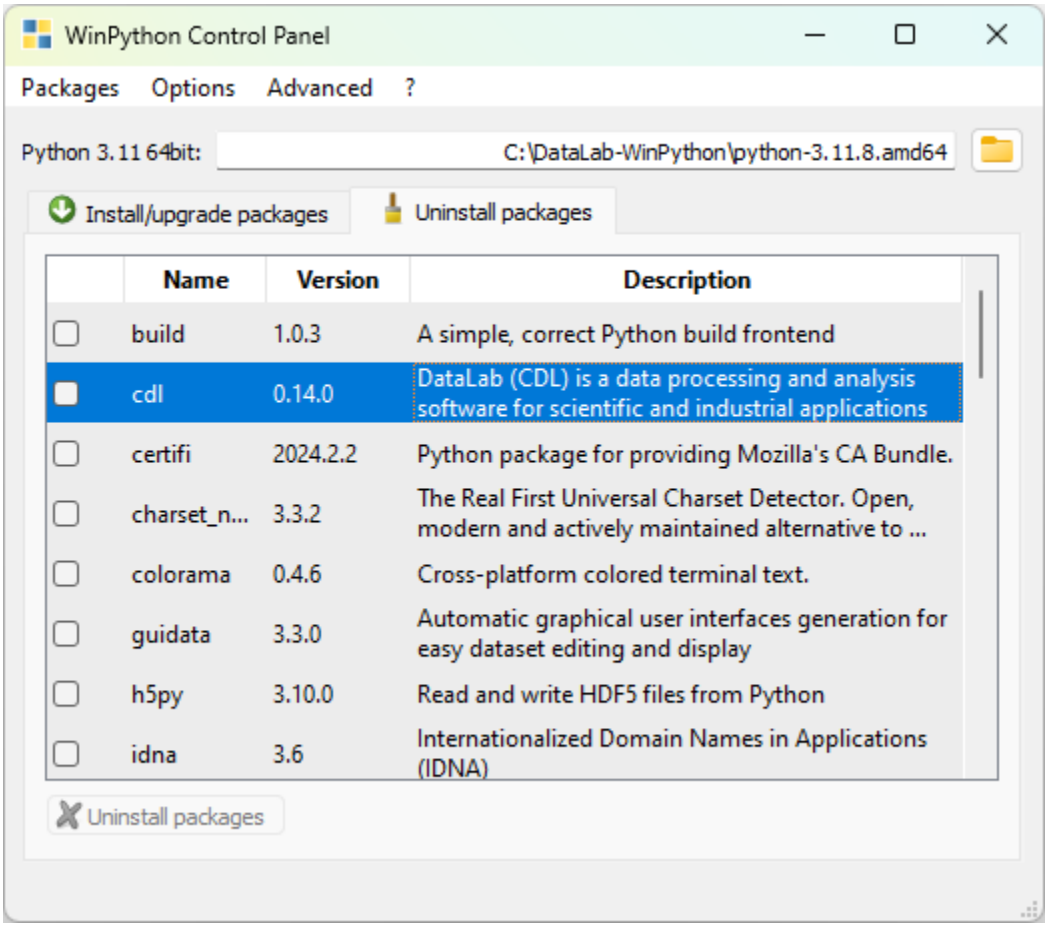


Fig. 3: DataLab-WinPython Control Panel

Warning: Whereas the all-in-one installer provides a monolithic package that guarantees the compatibility of all its components because it cannot be modified by the user, the WinPython distribution is more flexible and thus can be broken by a bad manipulation of the Python distribution by the user. This should be taken into account when choosing the installation method.

Wheel package

GNU/Linux Windows macOS

On any operating system, using pip and the Wheel package is the easiest way to install DataLab on an existing Python distribution:

```
$ pip install --upgrade DataLab-0.11.1-py2.py3-none-any.whl
```

Source package

GNU/Linux Windows macOS

Installing DataLab directly from the source package may be done using `pip`:

```
$ pip install --upgrade datalab-0.11.1.tar.gz
```

Or, if you prefer, you can install it manually by running the following command from the root directory of the source package:

```
$ pip install --upgrade .
```

Finally, you can also build your own Wheel package and install it using `pip`, by running the following command from the root directory of the source package (this requires the `build` and `wheel` packages to be installed):

```
$ pip install build wheel  # Install build and wheel packages (if needed)
$ python -m build  # Build the wheel package
$ pip install --upgrade dist/datalab-0.11.1-py2.py3-none-any.whl  # Install the wheel␣
↪package
```

Development version

GNU/Linux Windows macOS

If you want to try the latest development version of DataLab, you can install it directly from the master branch of the Git repository.

The first time you install DataLab from the Git repository, enter the following command:

```
$ pip install git+https://github.com/DataLab-Platform/DataLab.git
```

Then, if at some point you want to upgrade to the latest version of DataLab, just run the same command with options to force the reinstall of the package without handling dependencies (because it would reinstall all dependencies):

```
$ pip install --force-reinstall --no-deps git+https://github.com/DataLab-Platform/
↪DataLab.git
```

Note: If dependencies have changed, you may need to execute the same command as above, but without the `--no-deps` option.

1.1.2 Dependencies

Note: The DataLab all-in-one installer already include all those required libraries as well as Python itself.

The *datalab-platform* package requires the following Python modules:

Name	Version	Summary
Python	>=3.9, <4	Python programming language
guidata	>= 3.13.3	Automatic GUI generation for easy dataset editing and display
PlotPy	>= 2.8.2	Curve and image plotting tools for Python/Qt applications
Sigima	>= 1.0.2	Scientific computing engine for 1D signals and 2D images, part of the DataLab open-source platform.
NumPy	>= 1.22	Fundamental package for array computing in Python
SciPy	>= 1.10.1	Fundamental algorithms for scientific computing in Python
scikit-image	>= 0.19.2	Image processing in Python
pandas	>= 1.4	Powerful data structures for data analysis, time series, and statistics
PyWavelets	>= 1.2	PyWavelets, wavelet transform module
psutil	>= 5.8	Cross-platform lib for process and system monitoring.
packaging	>= 21.3	Core utilities for Python packages

Optional modules for GUI support (Qt):

Name	Version	Summary
PyQt5	>= 5.15.6	Python bindings for the Qt cross platform application toolkit

Optional modules for development:

Name	Version	Summary
build		A simple, correct Python build frontend
babel		Internationalization utilities
Coverage		Code coverage measurement for Python
pylint		python code static checker
ruff		An extremely fast Python linter and code formatter, written in Rust.
pre-commit		A framework for managing and maintaining multi-language pre-commit hooks.

Optional modules for building the documentation:

Name	Version	Summary
sphinx		Python documentation generator
sphinx_intl		Sphinx utility that make it easy to translate and to apply translation.
sphinx-sitemap		Sitemap generator for Sphinx
myst_parser		An extended [CommonMark](https://spec.commonmark.org/) compliant parser,
sphinx_design		A sphinx extension for designing beautiful, view size responsive web components.
sphinx-copybutton		Add a copy button to each of your code cells.
pydata-sphinx-theme		Bootstrap-based Sphinx theme from the PyData community

Optional modules for running test suite:

Name	Version	Summary
pytest		pytest: simple powerful testing with Python
pytest-xvfb		A pytest plugin to run Xvfb (or Xephyr/Xvnc) for tests.

Note: Python 3.11 and PyQt5 are the reference for production release

1.2 Use cases, main features and key strengths

DataLab is a platform for data processing and visualization (signals or images) that includes many functions. Developed in Python, it benefits from the richness of the associated ecosystem in terms of scientific and technical libraries.

1.2.1 What are the applications for DataLab?

Real world examples

A few concrete and specific examples illustrate the nature of the work that can be carried out with DataLab:

- Processing of experimental data (signals and images) acquired on a scientific facility in the nuclear field
- Processing of data acquired by a sensor in an industrial context
- Processing of images acquired by a camera in a medical context
- Automatic detection of defects on a surface, in the context of quality control
- Automatic detection of laser spots on a target, in the context of laser alignment
- Instrument alignment through image processing
- Automatic pattern detection on images and geometric correction of the images, in the context of non destructive testing

Usage modes

Depending on the application, DataLab can be used in three different modes:

- **Stand-alone mode**: DataLab is a full-fledged processing application that can be adapted to the client's needs through the addition of industry-specific plugins.
- **Embedded mode**: DataLab is integrated into your application to provide the necessary processing and visualization features.
- **Remote-controlled mode**: DataLab communicates with your application, allowing it to benefit from its functionality without disrupting the user experience.

Use cases

See also:

For practical examples of use cases, see the *Tutorials* section:

- Most of the tutorials are describing concrete examples of use of DataLab in a scientific or technical context.
- Regarding the use of DataLab with an IDE (Integrated Development Environment) such as Visual Studio Code or Spyder, see the tutorial *DataLab and Spyder: a perfect match*.
- As for the use of DataLab with Jupyter notebooks, that is one of the topics covered in the tutorial *Add your own features*.

DataLab is a versatile tool that can be used in different contexts:

Data processing
DataLab is a powerful tool for processing signals and images. It can be used to develop complex algorithms, or to quickly prototype a processing chain.

See our *Tutorials* for practical examples of use in data processing.

Companion tool for scientific/technical work
DataLab can be used as a companion tool for scientific/technical work. It allows you to visualize and process data, and to share your results with your colleagues. It can easily be adapted to your needs through the addition of plugins, and it may even be used together with your every day tools (e.g., Visual Studio Code, Spyder… or Jupyter notebooks).

See our *Tutorials* for practical examples of use in a scientific/technical context.

Prototyping a data processing application
DataLab can be used to quickly prototype a data processing application. It can then be used as a basis for the development of a full-fledged application.

See the tutorial *Prototyping a custom processing pipeline* for a concrete example.

Debugging a data processing application
DataLab can be used as an advanced debugging tool for your data processing applications, independently from the development environment or the language used (Python, C#, C++, etc.). All you need is to be able to communicate with DataLab via its remote control interface (standard XML-RPC protocol). This allows you to send data to DataLab (signals, images or even geometric shapes), visualize the data at each step of the processing chain, manipulate them to better understand the behavior of your algorithms, and even modify them to test the robustness of your code.

See the tutorial *Debugging your algorithm with DataLab* for a quick overview of this feature.

Note: DataLab can also be controlled from your familiar development environment (e.g., Visual Studio Code, Spyder…) or from a Jupyter notebook, in order to perform calculations using your processing functions while leveraging

the advanced features of DataLab. See the tutorials *Prototyping a custom processing pipeline* or *DataLab and Spyder: a perfect match* for examples of use.

With its user-friendly experience and versatile usage modes, DataLab enables efficient development of your data processing and visualization applications while benefiting from an industrial-grade technological platform.

1.2.2 Main features

The main technical features of DataLab include:

- Support for numerous standard and proprietary data formats
- Opening an arbitrary number of objects (signals or images) for batch processing, with the possibility of defining groups of objects
- Simultaneous viewing of multiple objects with annotation support
- Standard operations and processing on signals and images
- Advanced image processing (restoration, morphology, edge detection, etc.)
- Management of multiple regions of interest (calculations, extractions)
- Macro-command editor
- Remote-controllable API
- Embedded interactive Python console

1.2.3 Key strengths

To summarize, the four key strengths of DataLab are:

Extensibility
: The DataLab plugin system makes it easy to code new features (specific processing, specific file formats, custom graphical interfaces). It can also be used as a customizable platform.

Interoperability
: DataLab can also be embedded in your own application. For example, within data processing software, machine-level control systems, or test bench applications.

Automation
: A high-level public API allows for full remote control of DataLab to open and process data.

Maintainability and testability
: DataLab is an industrial-grade scientific and technical processing software. The built-in automated tests in DataLab cover 90% of its features, which is significant for software with graphical interfaces and helps mitigate regression risks. Moreover, the test suite includes validation tests based either on ground truth data or analytical solutions.

See also:

See section *Overview & Common features* for more information on DataLab's validation strategy.

1.3 Key features

This page presents briefly DataLab key features.

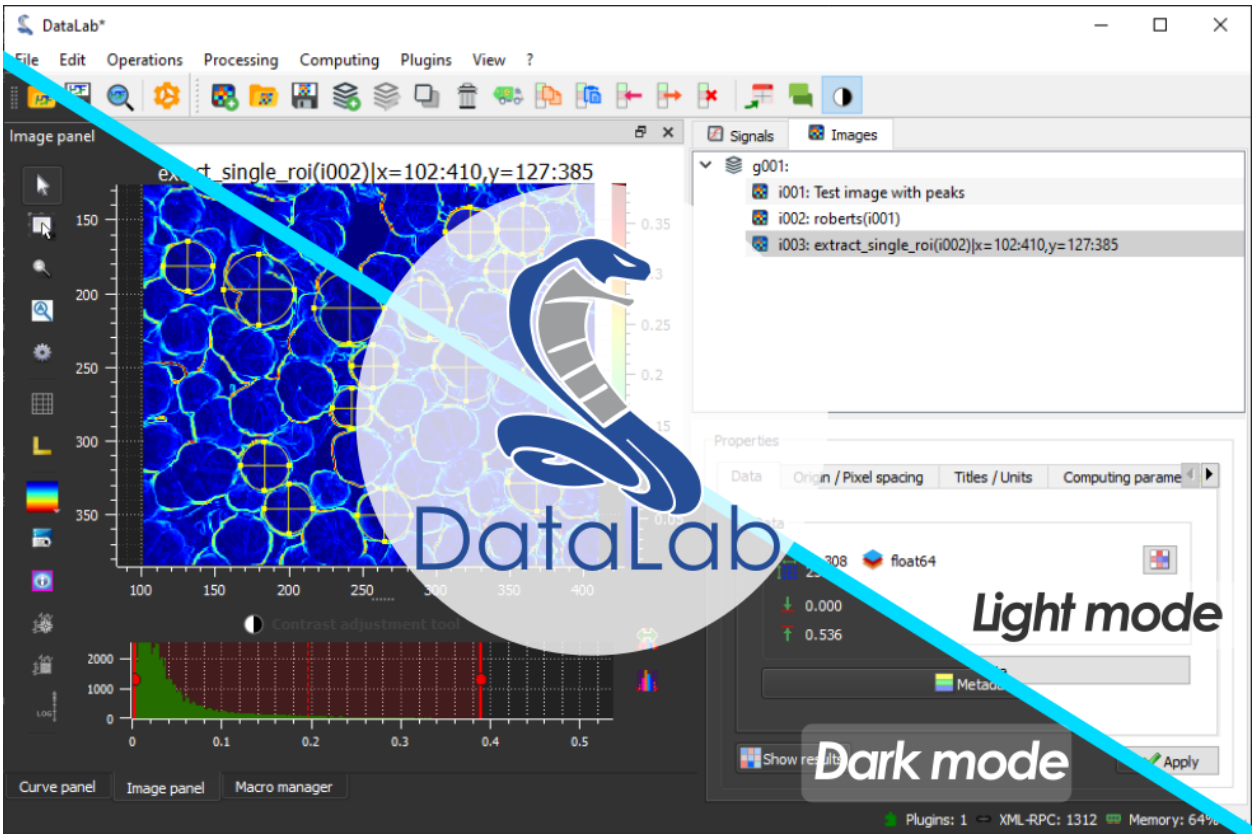


Fig. 4: DataLab supports dark and light mode depending on your platform settings (this is handled by the guidata package, and may be overridden by setting the *QT_COLOR_MODE* environment variable to *dark* or *light*).

1.3.1 Data visualization

Signal	Image	Feature
✓	✓	Screenshots (save, copy)
✓	Z-axis	Lin/log scales
✓	✓	Data table editing
✓	✓	Statistics on user-defined ROI
✓	✓	Markers
	✓	Aspect ratio (1:1, custom)
	✓	50+ available colormaps (customizable)
	✓	Intensity profiles (line, average, radial)
✓	✓	Annotations
✓	✓	Persistance of settings in workspace
	✓	Distribute images on a grid
✓	✓	Single or superimposed views

1.3.2 Data processing

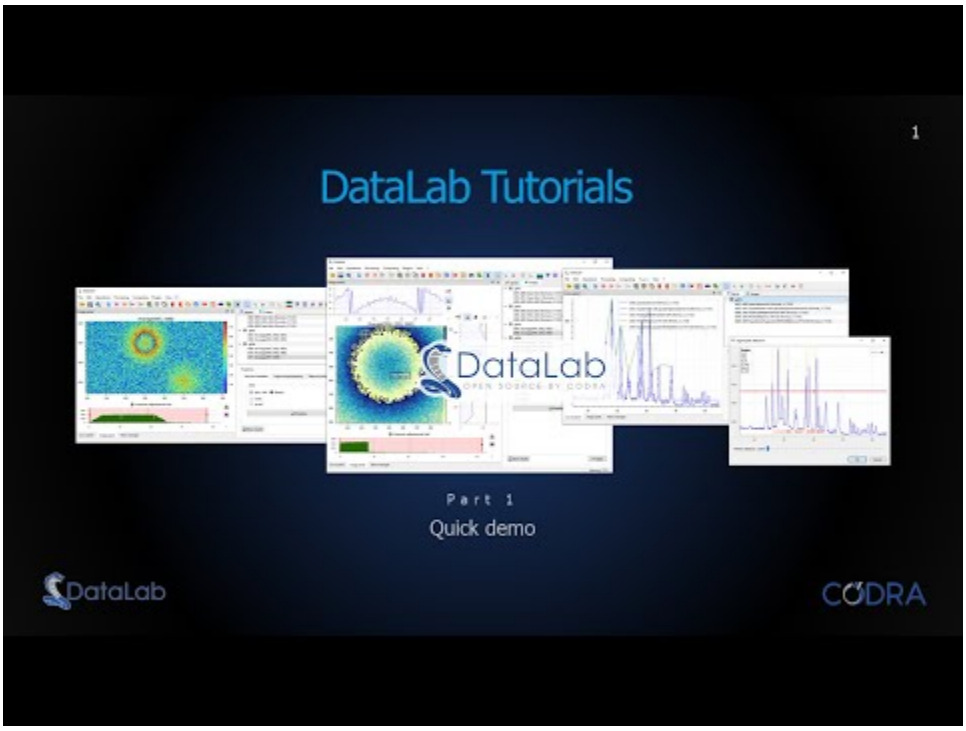
Signal	Image	Feature
✓	✓	Process isolation for running computations
✓	✓	Remote control from Jupyter, Spyder or any IDE
✓	✓	Remote control from a third-party application
✓	✓	Sum, average, difference, product…
✓	✓	Operations with a constant
✓	✓	Square root, power, logarithm, exponential…
✓	✓	Average, standard deviation
✓		Derivative, integral
✓	✓	ROI extraction, Swap X/Y axes
✓		Semi-automatic multi-peak detection
✓	✓	Convolution, deconvolution
	✓	Flat-field correction
	✓	Flip, rotation, scaling…
	✓	Intensity profiles (line, average, radial)
	✓	Pixel binning
✓	✓	Linear calibration
✓	✓	Normalization, Clipping, Offset correction
✓		Reverse X-axis
	✓	Thresholding (manual, Otsu…)
✓	✓	Gaussian filter, Wiener filter
✓	✓	Moving average, moving median
✓	✓	FFT, inverse FFT, Power/Phase/Magnitude spectrum, Power Spectral Density
✓		Interpolation, resampling
✓		Detrending
✓		X-Y mode
✓		Interactive fit: Gauss, Lorentz, Voigt, polynomial, CDF…
✓		Interactive multigaussian fit
✓		Frequency filters (low-pass, high-pass, band-pass, band-stop)
✓		Windowing (Hann, Hamming…)
	✓	Butterworth filter
	✓	Exposure correction (gamma, log…)
	✓	Restauration (Total Variation, Bilateral…)
	✓	Morphology (erosion, dilation…)
	✓	Edges detection (Roberts, Sobel…)
✓	✓	Analysis on custom ROI
✓		FWHM, FW @ $1/e^2$
✓		Dynamic parameters (ENOB, SNR…), Sampling period/Rate
	✓	Centroid (robust method w/r noise)
	✓	Minimum enclosing circle center
	✓	2D peak detection
	✓	Contour detection
	✓	Circle Hough transform
	✓	Blob detection (OpenCV, Laplacian of Gaussian…)

1.4 Tutorials

1.4.1 Video Tutorials

DataLab video tutorials intend to provide a complementary material to the documentation and other tutorials available here.

Quick demo

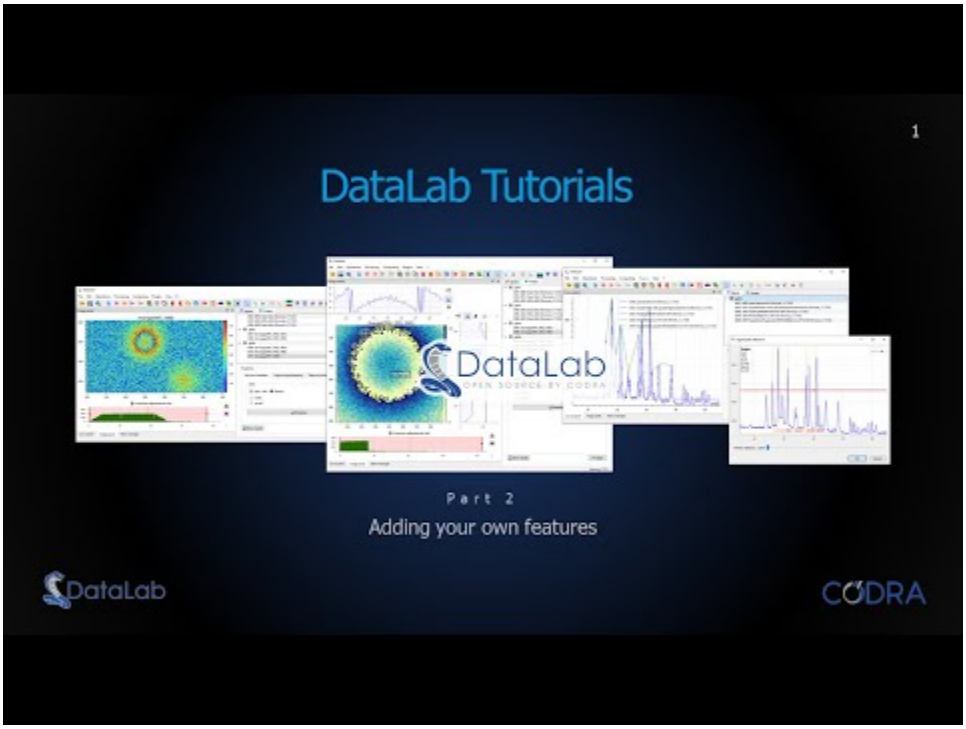


Warning: This video shows an older version of DataLab. Some features may have changed since then. Please refer to the documentation for the most up-to-date information.

In this first video, we will quickly go through the main features of DataLab, and show how to do some basic signal and image processing.

Note: This video is intentionally short and does not cover all the features of DataLab. It is meant to give you a quick overview of the software. For a more detailed description of the features, please watch the other videos or read the documentation.

Add your own features



Warning: This video shows an older version of DataLab. Some features may have changed since then. Please refer to the documentation for the most up-to-date information.

In this tutorial, we will show how to add your own features to DataLab using three different approaches:

1. Macro-commands, using the integrated macro manager
2. Remote control of DataLab from an external IDE (e.g. Spyder) or a Jupyter notebook
3. Plugins

The first common point between these three approaches is that they all rely on DataLab's high-level API, which allow to interact with almost every aspect of the software. This API is here accessed using Python scripts, but it may also be accessed using any other language when using the remote control approach (because it relies on a standard communication protocol, XML-RPC).

The second common point is that they all use Python code, and compatible proxy objects, so that the same code can be at least partially reused in the three approaches.

1.4.2 Other Tutorials

The following tutorials are detailed step-by-step guides to perform specific tasks with DataLab, or to illustrate features of the software in the context of a scientific or technical problem. Each tutorial focuses on a specific aspect of the software and is intended to be self-contained.

Processing a spectrum

This example shows how to process a spectrum with DataLab:

- Read the spectrum from a file
- Apply a filter to the spectrum
- Extract a region of interest
- Fit a model to the spectrum
- Save the workspace to a file

DataLab menus change depending on the context. Since we are going to work with a spectrum, a 1D signal, we need to select the Signal Panel before proceeding.

In a typical workflow, we would open DataLab and read the spectrum from a file using "File > Open…", the button in the toolbar, or by dragging and dropping the file into the panel on the right. However, for this tutorial, we will use the "Plugins > Test data > Load spectrum of paracetamol" feature to generate a test spectrum, which is convenient for demonstration purposes.

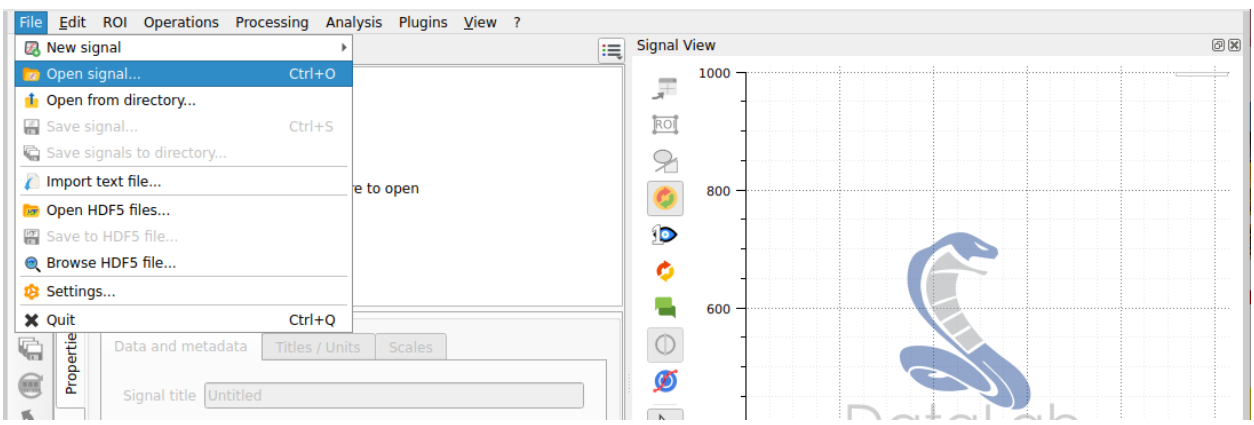


Fig. 5: The "File > Open…" menu to open a spectrum file.

Once opened, the spectrum is displayed in the main window. It is a 1D signal, so it is displayed as a curve. The horizontal axis is the energy axis, and the vertical axis is the intensity axis.

The signal is quite clean. However, to demonstrate DataLab's filtering capabilities, we will apply a Wiener filter to reduce any residual noise while preserving the spectral features. This is available under "Processing > Noise Reduction > Wiener filter".

Let's focus our analysis on one of the peaks of interest. To do that, we define a region of interest (ROI) around the feature we want to analyze and use the "ROI > Extract…" menu to extract it. The "Regions of interest" dialog box will be displayed. Select an area and click "OK". A confirmation window will appear—click "Yes" to extract the region of interest. A new signal containing the ROI will be created and displayed in the main window.

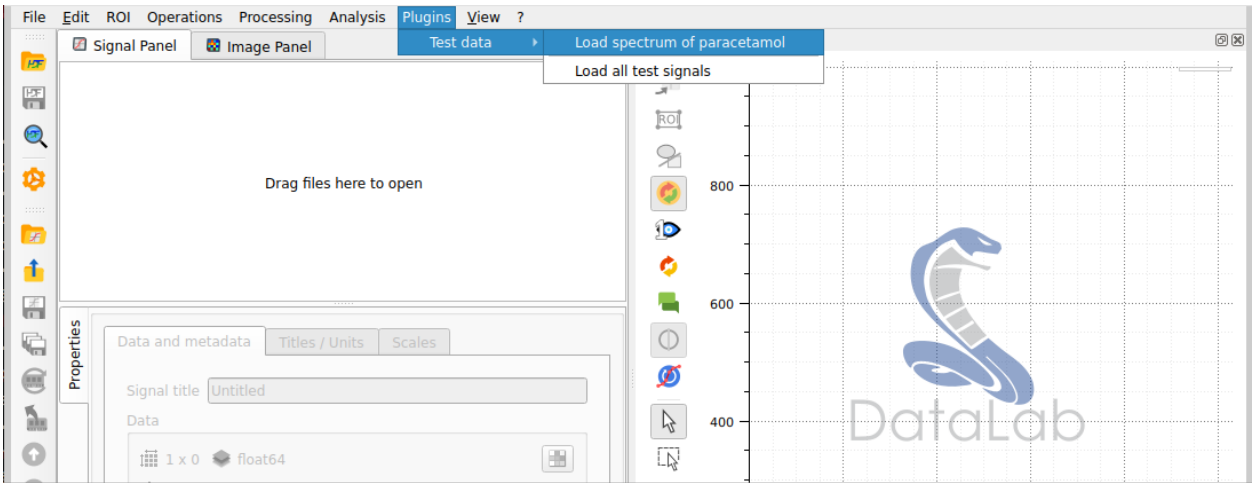


Fig. 6: The "Plugins > Test data > Load spectrum of paracetamol" plugin to generate the test spectrum for this tutorial.

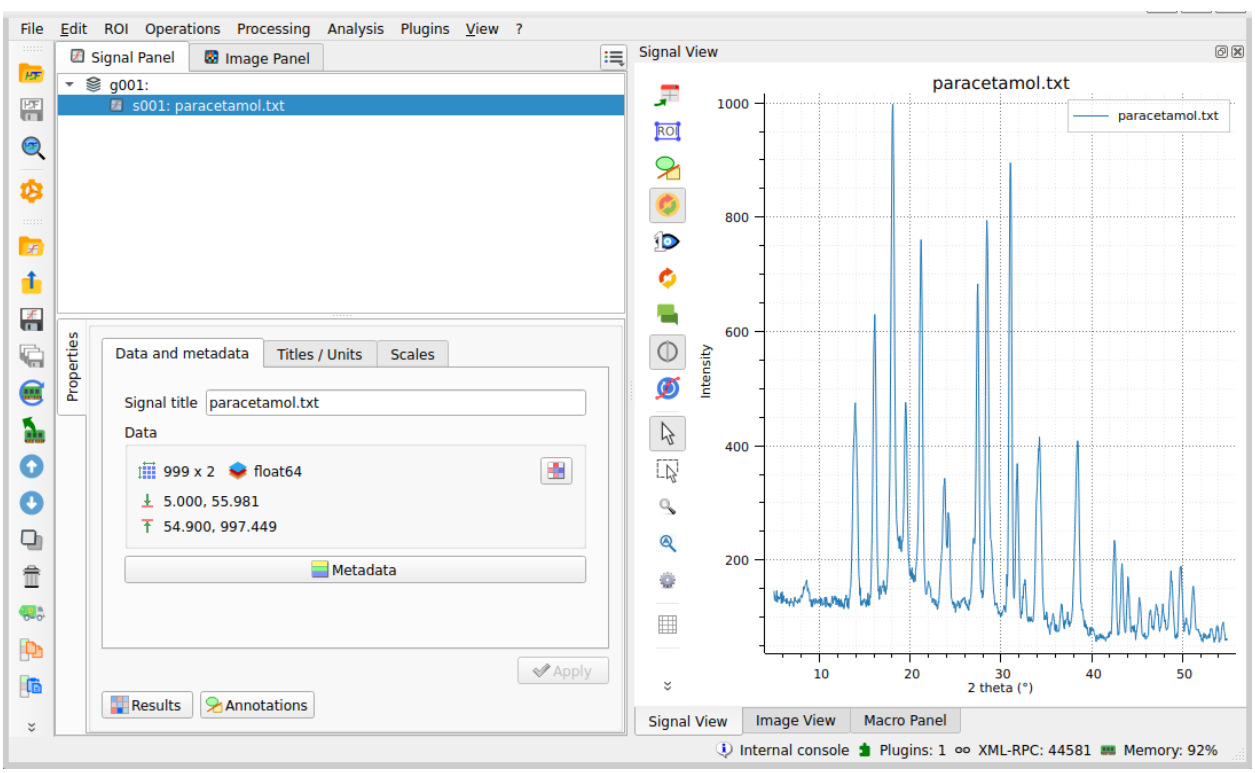


Fig. 7: The spectrum displayed on the "Signal View" panel.

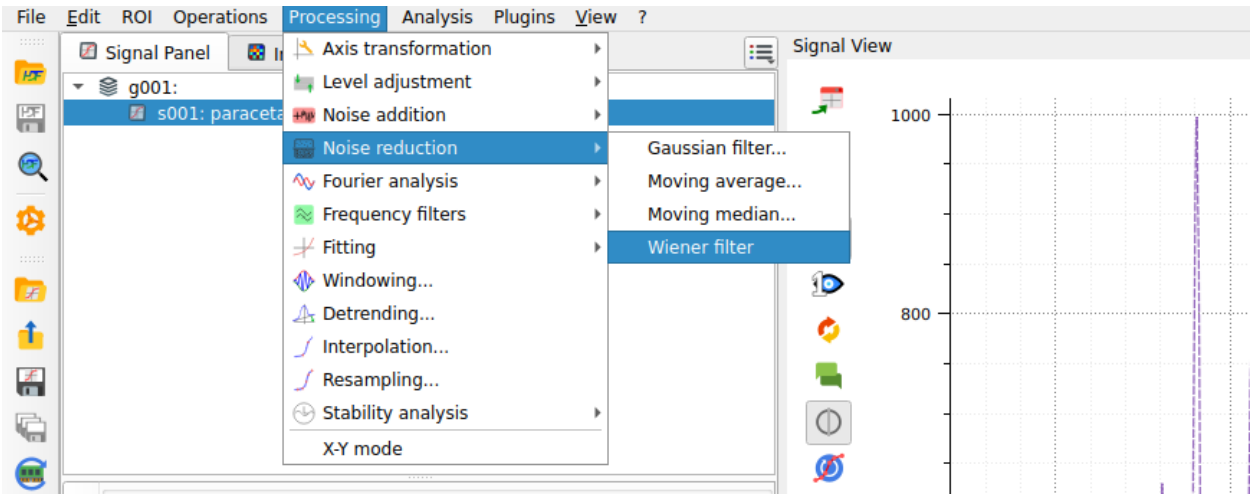


Fig. 8: The "Processing > Noise Reduction > Wiener filter" menu option.

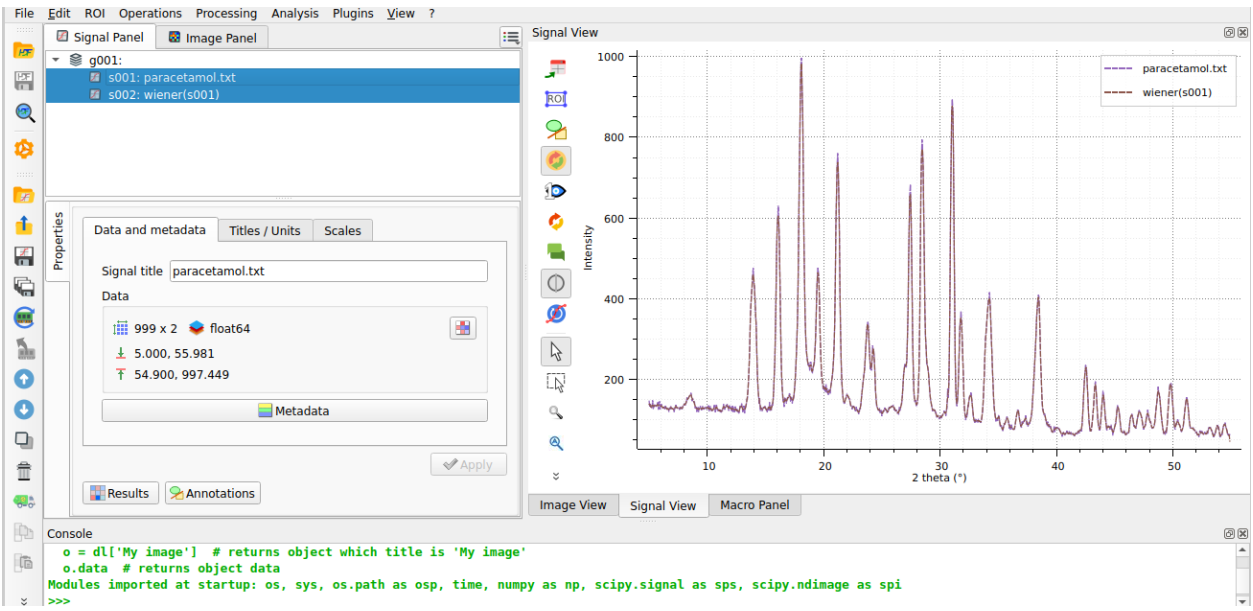


Fig. 9: The result displayed in the main window.

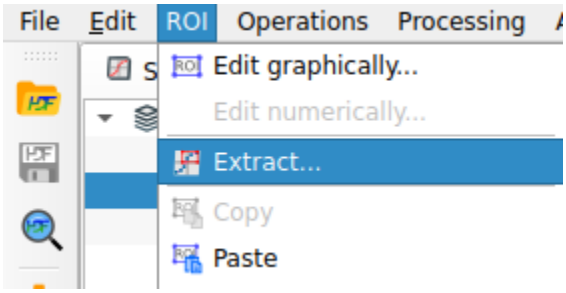


Fig. 10: The "Operations > ROI extraction" menu.

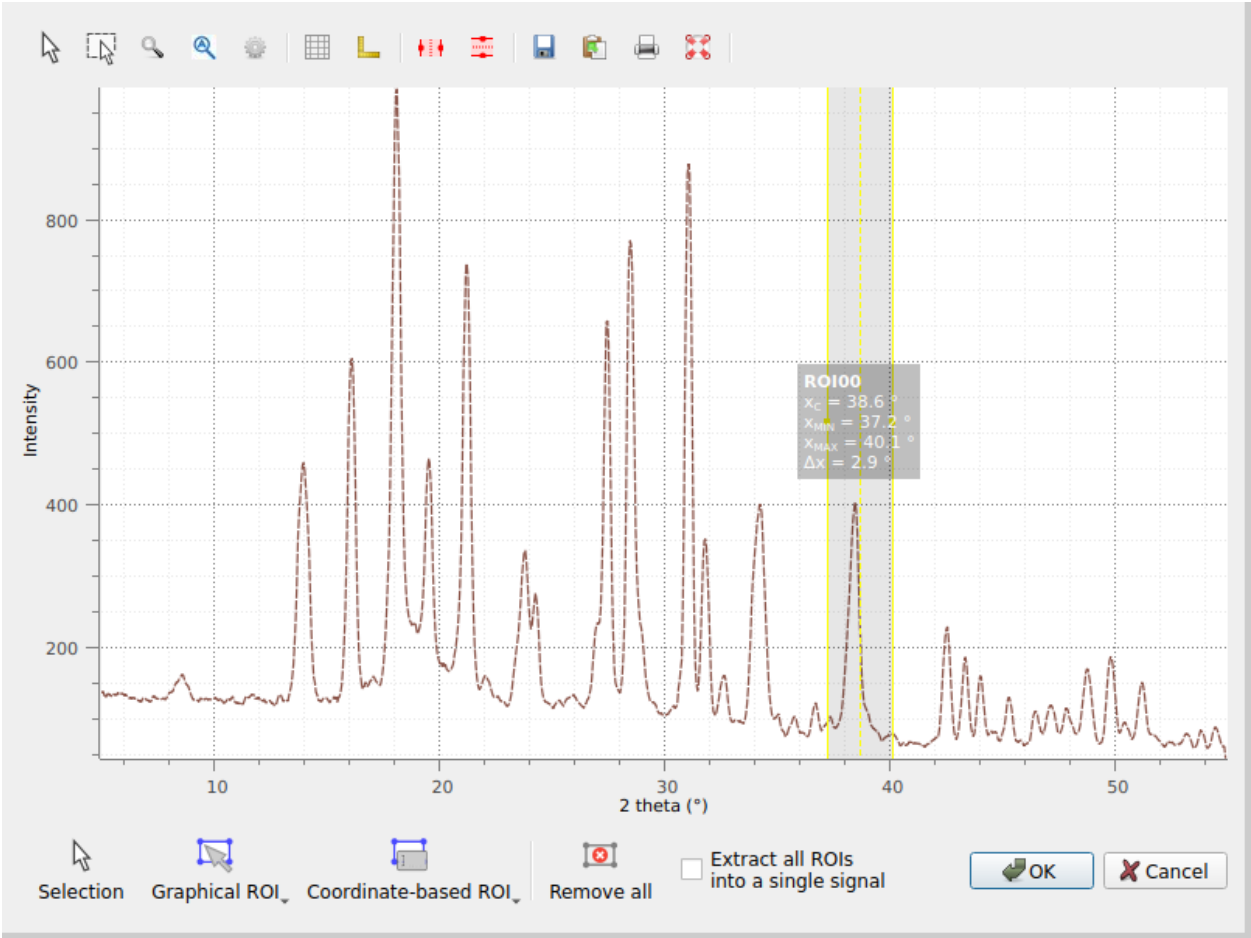


Fig. 11: The "Regions of interest" dialog box displayed.

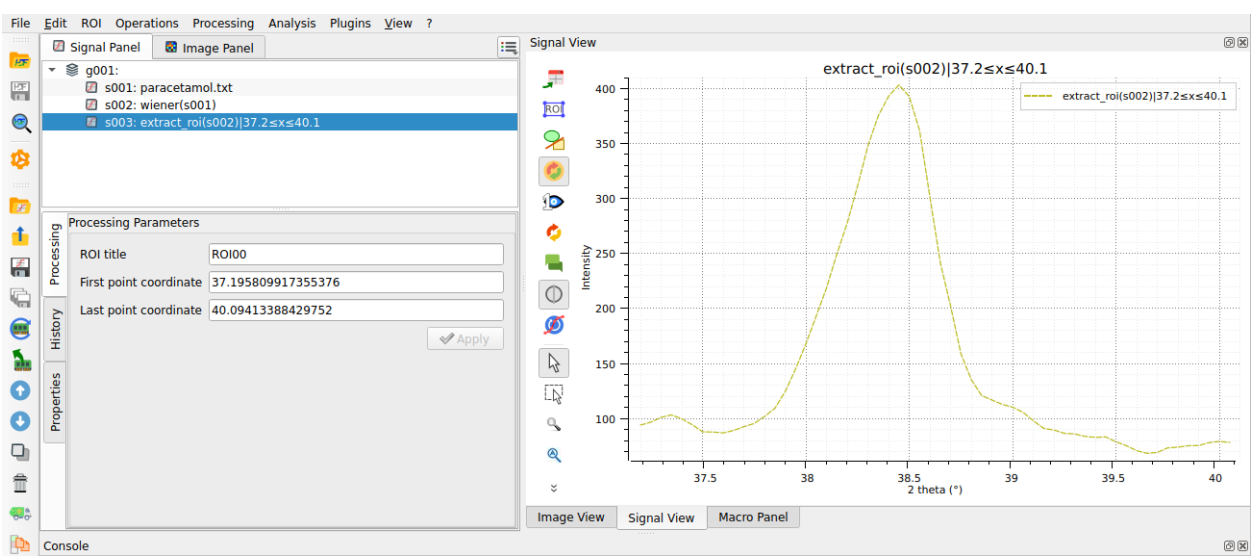


Fig. 12: The region of interest displayed in the main window.

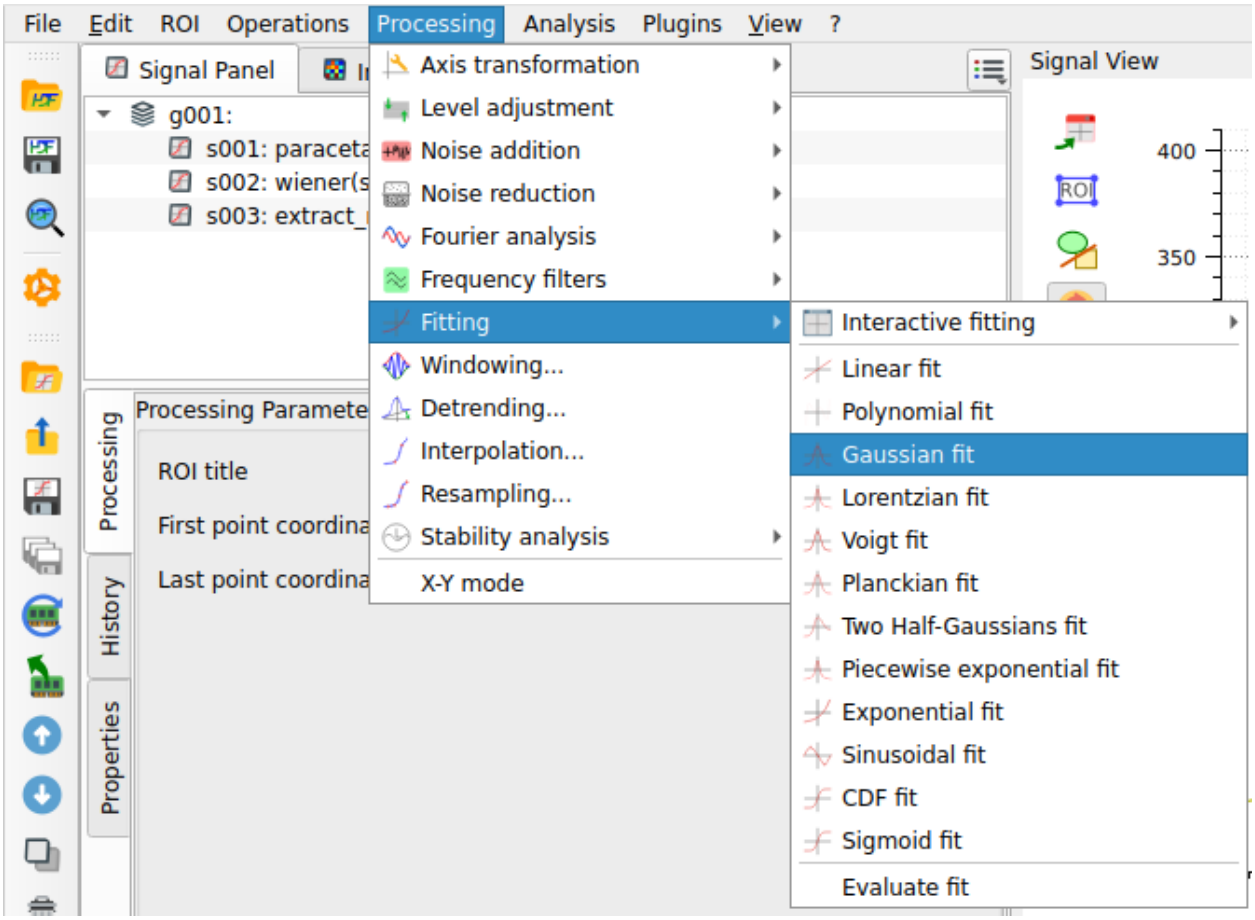


Fig. 13: Open the model fitting window with "Processing > Fitting > Gaussian fit".

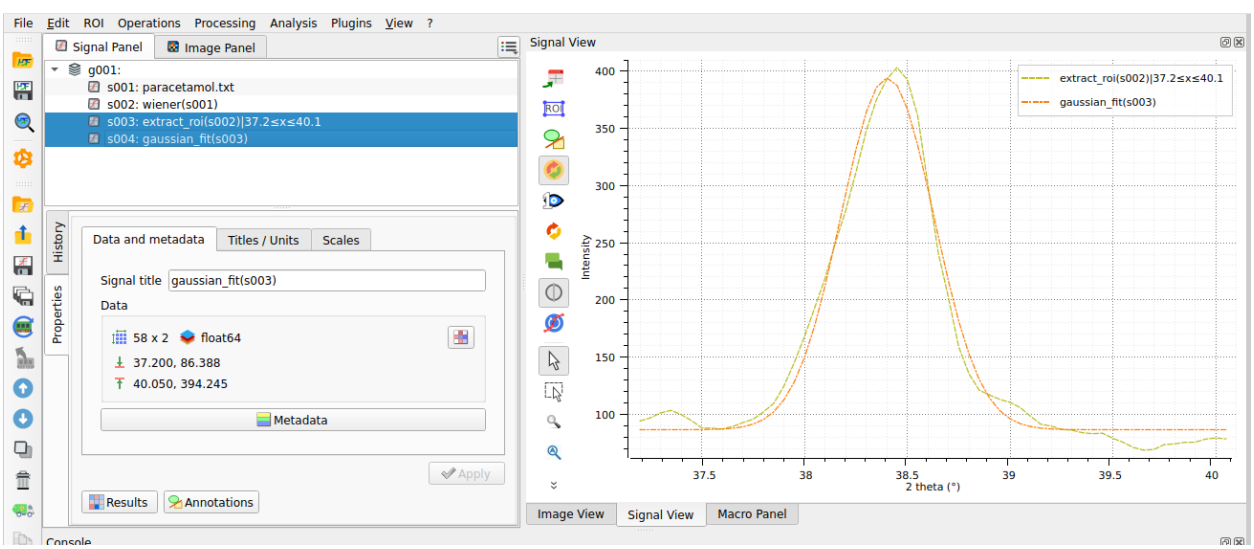


Fig. 14: The result of the fit displayed in the main window.

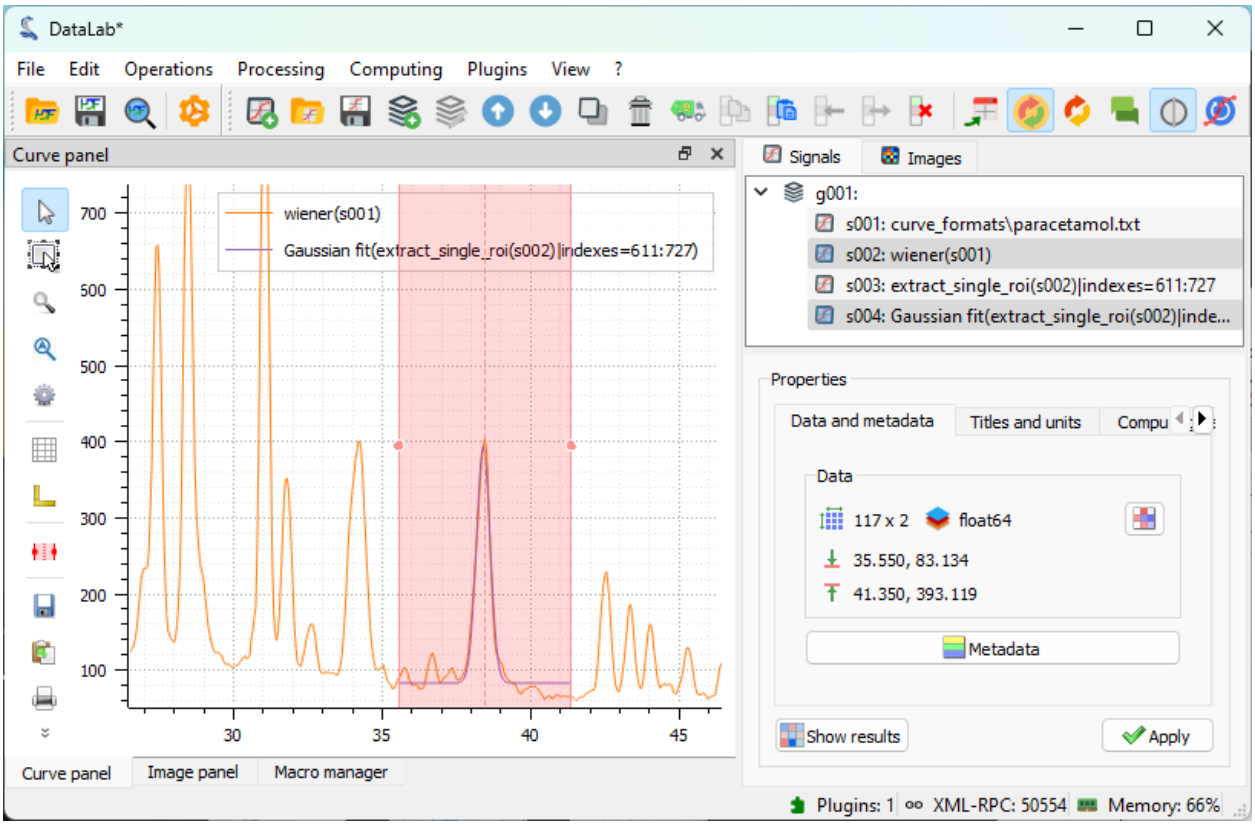


Fig. 15: Both the full spectrum and the fit are selected in the "Signals" panel, so that both are displayed in the visualization panel on the left. This allows for easy visual comparison if needed for the analysis.

Linear detrending

After fitting the main peak, we may want to remove any baseline drift present in the entire spectrum. The detrending function in DataLab performs a linear fit across the entire signal, including the peaks. In our signal, the peaks occupy a large portion of the data, which is acceptable for signals where peaks are symmetrically distributed around the center with similar amplitudes. However, this is not the case here, so we cannot expect this function to work well. Nevertheless, this example illustrates how DataLab functions can be combined to perform more advanced analyses.

To visualize the limitation mentioned above, we will apply the detrending function directly to the filtered signal. It's important to remember that we previously set a ROI on the signal to focus on the main peak. We need to remove this ROI constraint to perform detrending on the full signal. To execute the detrending, we use "Processing > Detrending", choose the linear detrending method, and click "OK". The result will be displayed in the main window.

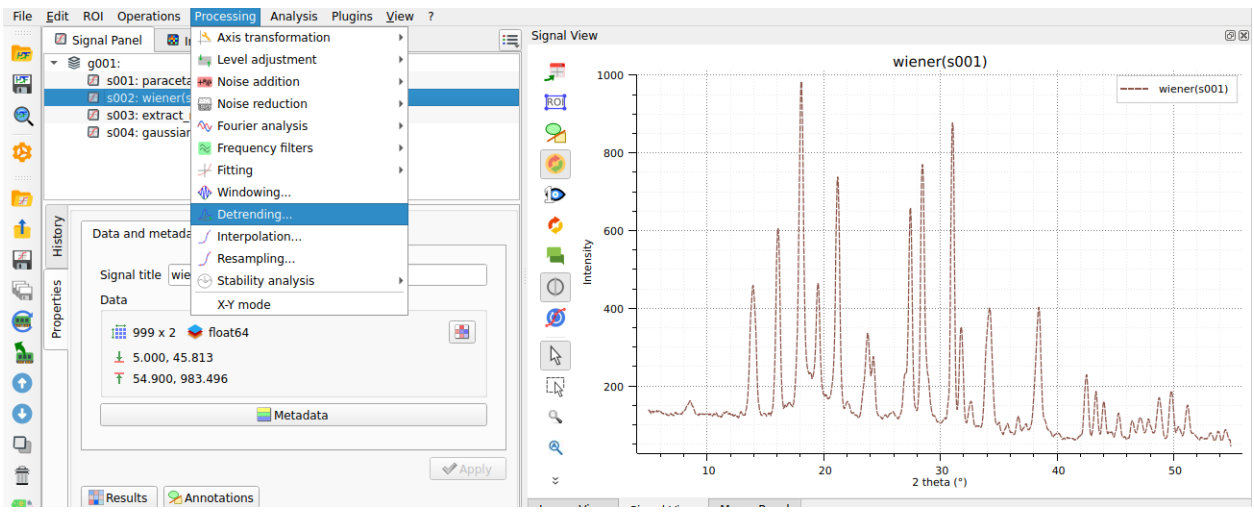


Fig. 16: The "Processing > Detrending" feature.

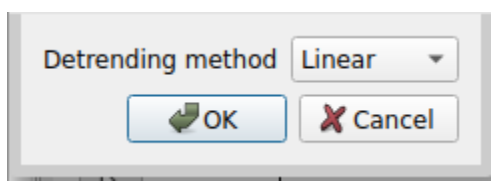


Fig. 17: We choose a linear detrending method, and we click on "OK".

Comparing the filtered and detrended signals shows, as expected, that the detrending function does not work well on this signal. As explained earlier, this is because the algorithm performs a linear fit across the entire signal, including the peaks. The effect is clearly visible in the plot: the higher peaks on the left start at an intensity value lower than those on the right after detrending, and all peaks have a baseline below zero.

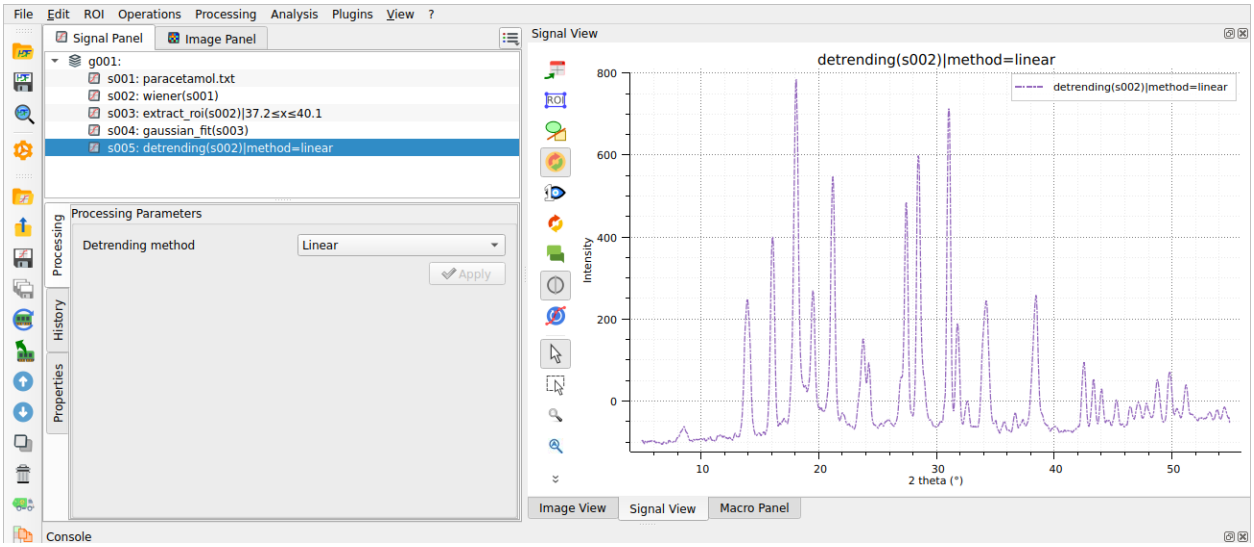


Fig. 18: The result of the detrending displayed in the main window.

Improved detrending with peak exclusion

To overcome the detrending function's limitation, we can draw inspiration from the behavior of the detrended signal. We've already identified the problem: the linear fit includes both the baseline and the peaks.

For better detrending, we can first exclude the peaks and then perform a linear fit only on the non-peak regions. We can reasonably expect this approach to provide a more accurate baseline estimation and a better detrended signal.

This is illustrated in the following steps:

Automatic peak detection

We can use DataLab's "Multi-Gaussian fit" function to automatically identify and fit multiple peaks in the spectrum by selecting "Processing > Fitting > Multi-Gaussian fit" from the menu.

Alternatively, we could use the "Peak detection" feature from the "Analysis" menu to detect peaks in the spectrum. This is the first step of the "Multi-Gaussian fit" function and can be used independently to detect peaks without performing a fit, creating a signal with a delta function at each detected peak position.

Saving the workspace

Finally, we can save the workspace to a file. The workspace contains all signals loaded or created in DataLab, along with processing results and visualization settings (such as curve colors).

If you want to load the workspace again, you can use the "File > Open HDF5 file…" (or the button in the toolbar) to load the whole workspace, or the "File > Browse HDF5 file…" (or the button in the toolbar) to load only a selection of data sets from the workspace.

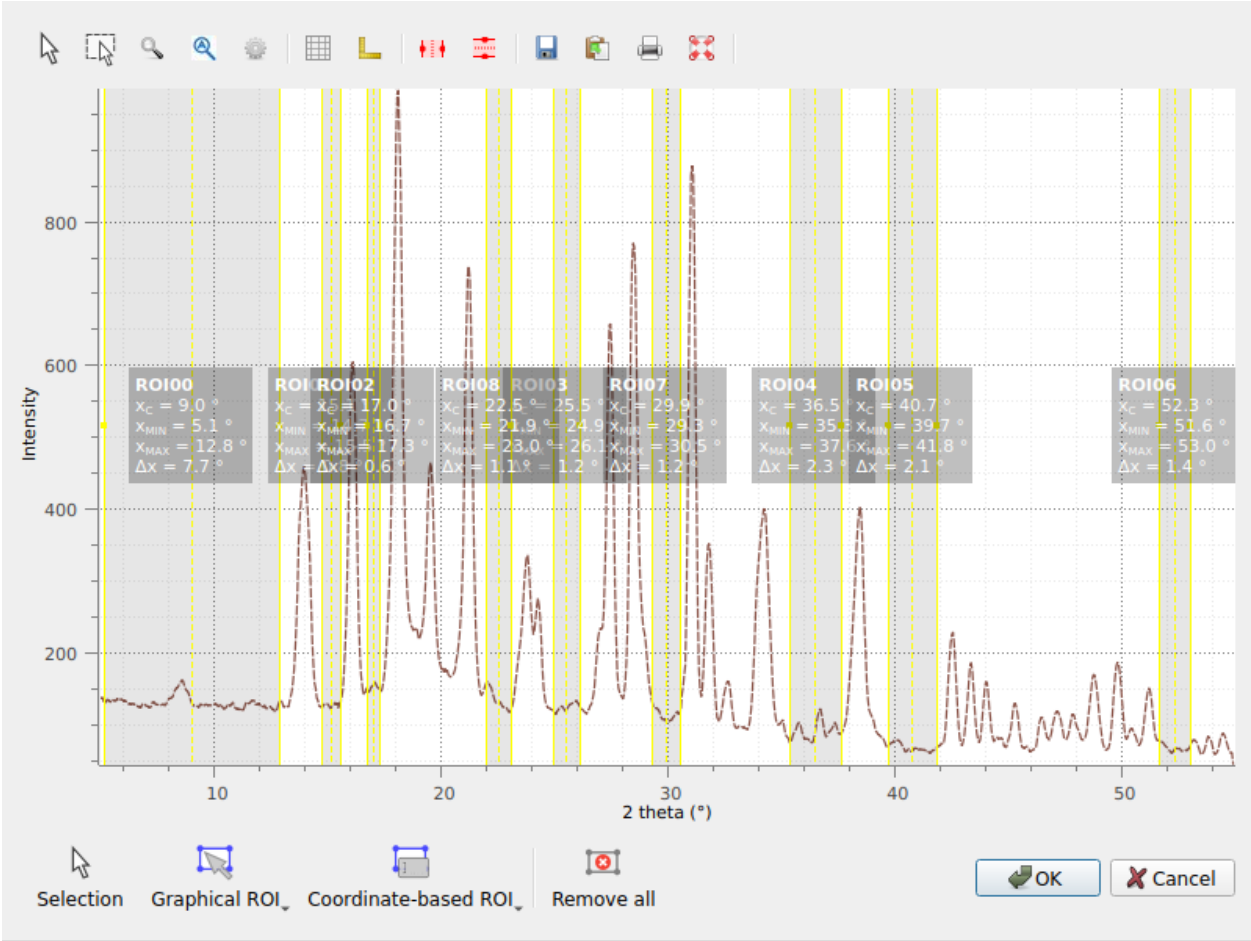


Fig. 19: We select the regions corresponding to the regions without peaks.

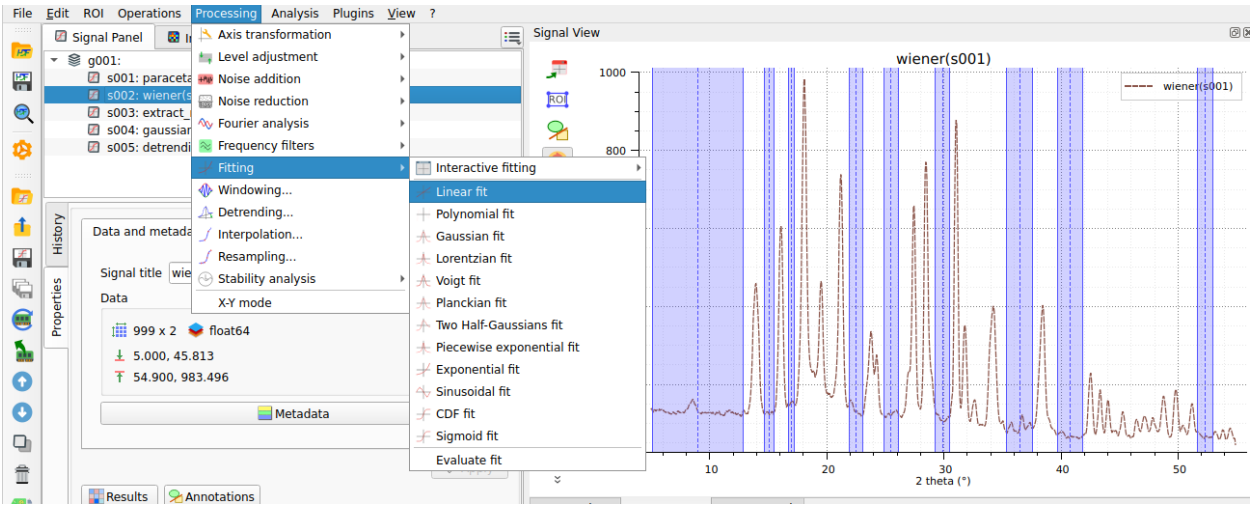


Fig. 20: A linear fit is performed only on the selected regions.

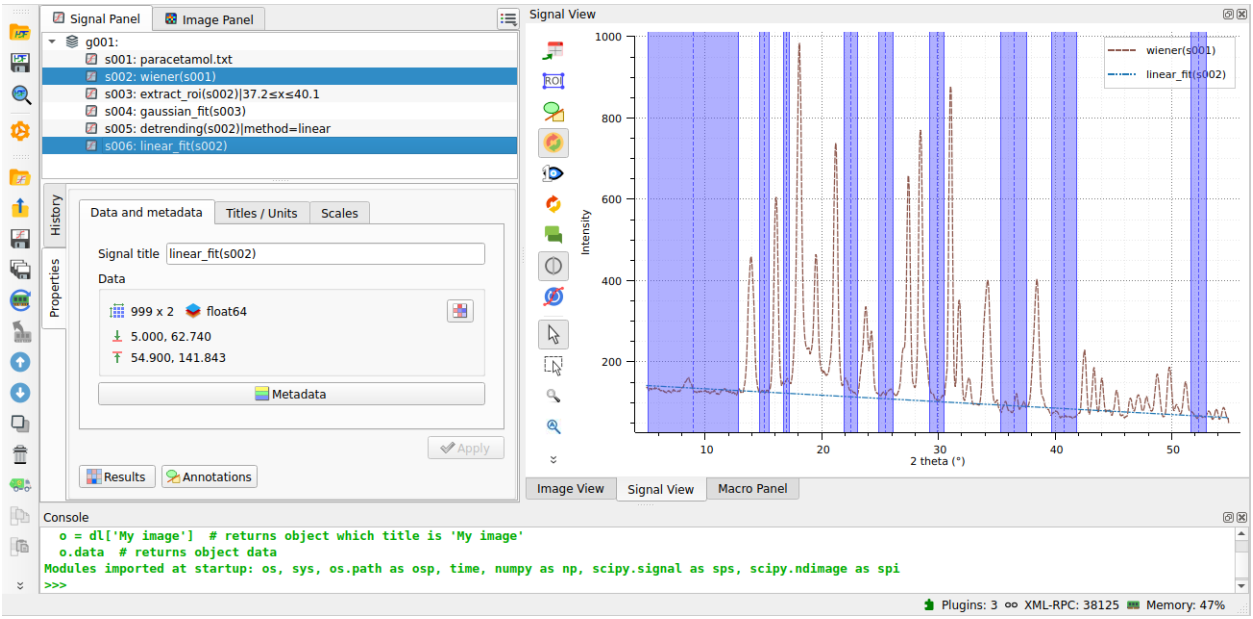


Fig. 21: The linear baseline obtained from the fit is shown.

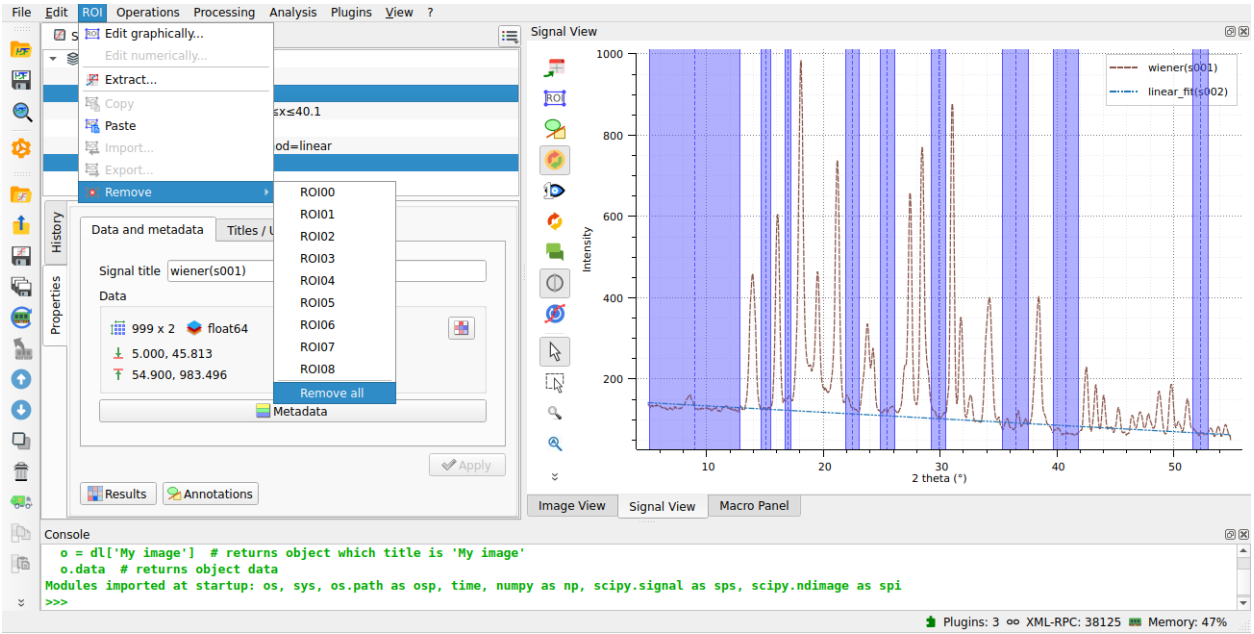


Fig. 22: We delete the rois to apply the detrending on the whole signal.

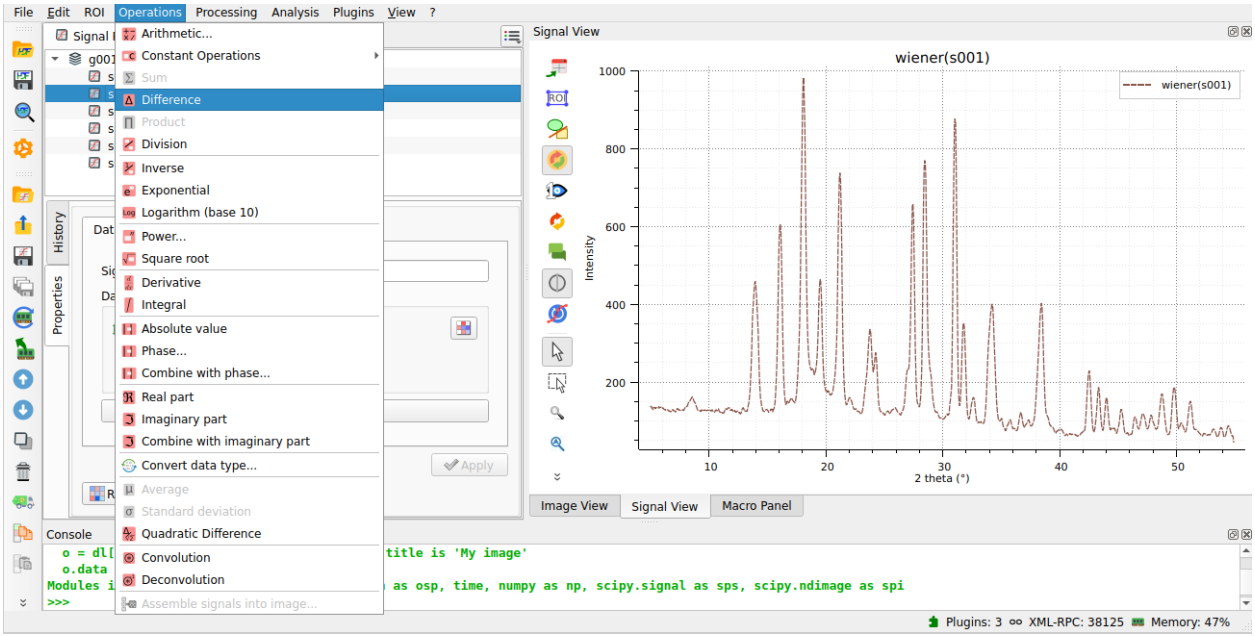


Fig. 23: We use the "difference" operation to subtract the baseline from the original signal.

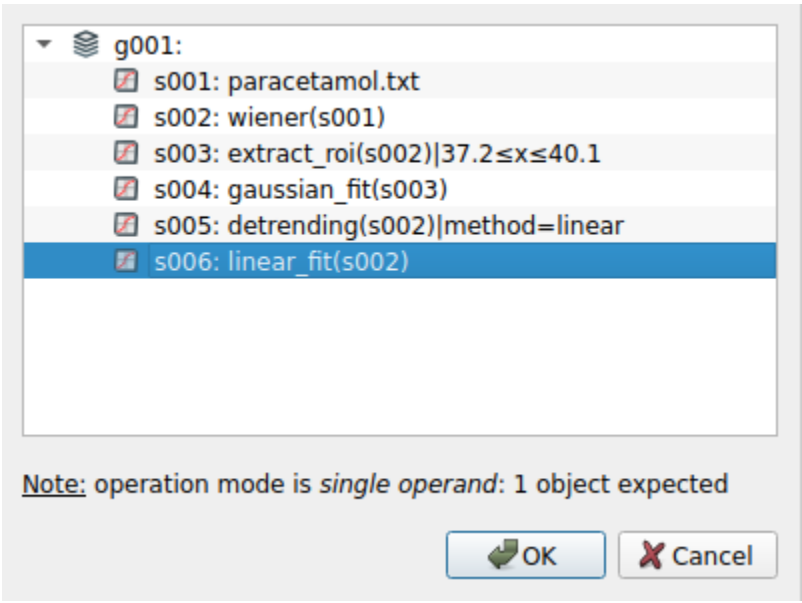


Fig. 24: We select the linear baseline to perform the subtraction.

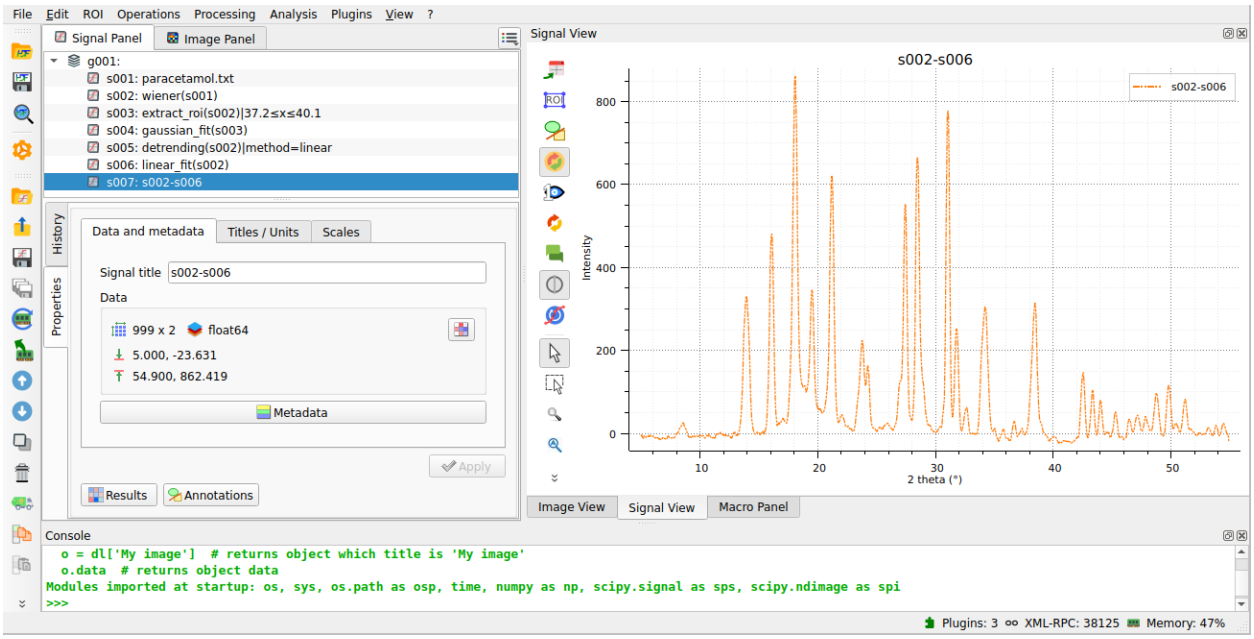


Fig. 25: The resulting detrended signal is shown, now with a correct baseline.

Detecting blobs on an image

This example shows how to detect blobs on an image with DataLab, and also covers other features such as the plugin system:

- Add a new plugin to DataLab
- Denoise an image
- Detect blobs on an image
- Save the workspace to a file

First, we open DataLab, and open the settings dialog (using "File > Settings…", or the icon in the toolbar).

See also:

The plugin system is described in the *Plugins* section.

Let's add the *datalab_example_imageproc.py* plugin to DataLab (this is an example plugin that is shipped with DataLab source package, or may be downloaded from here).

If we close and reopen DataLab, we can see that the plugin is now available in the "Plugins" menu: there is a new "Extract blobs (example)" entry.

Note: DataLab menus change between images and signals. To be able to see the "Extract blobs (example)" menu, make sure that the "Images" panel is selected (click on the "Images" tab on the left).

A pop-up dialog appears, asking for the parameters of the test image to generate (the size of the image and its title). We can just keep the default parameters and click on "OK".

For information, the image is generated by the plugin using the following code:

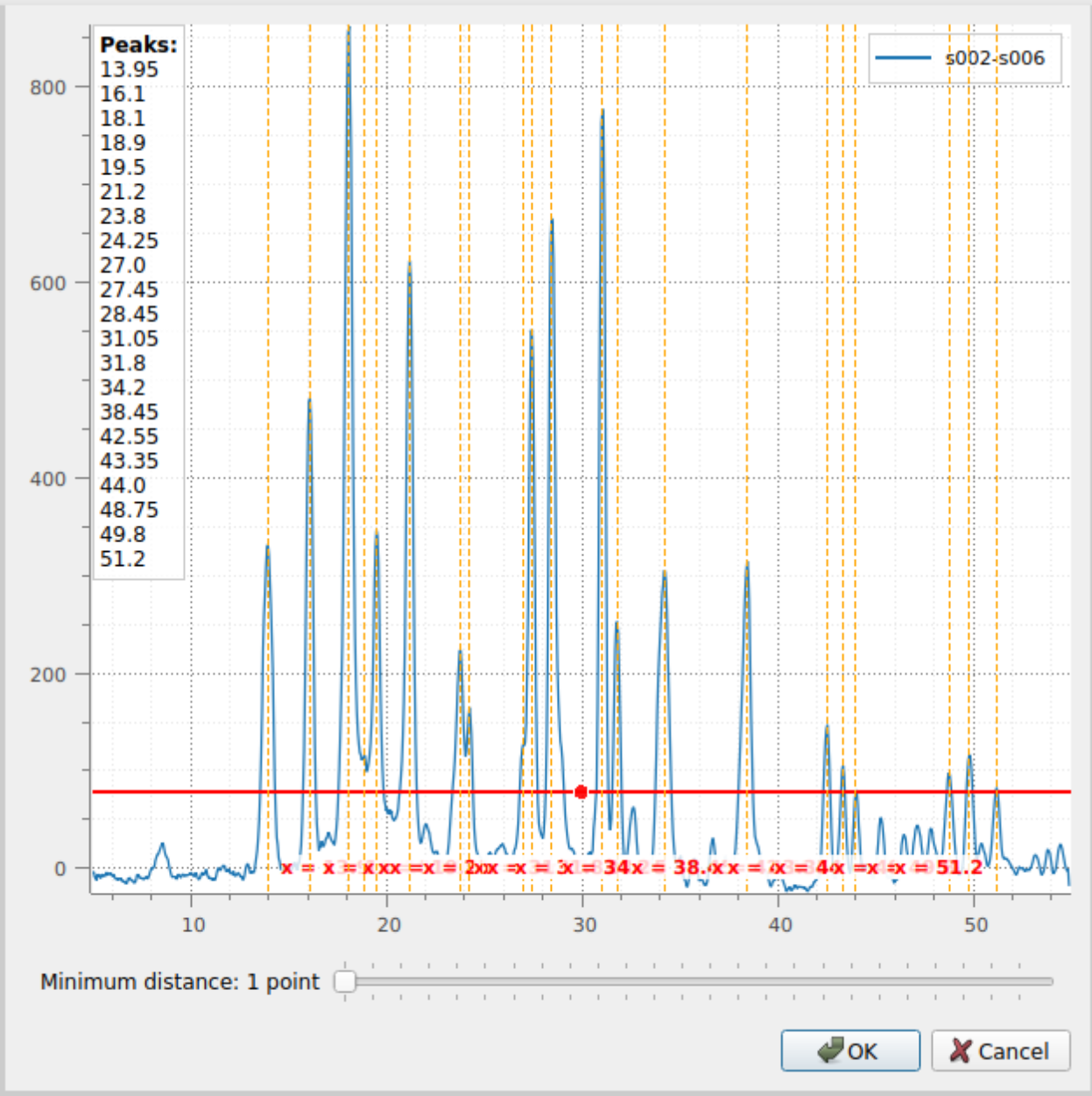


Fig. 26: First, the "Signal peak detection" dialog box appears. You can adjust the vertical cursor position to set the threshold for peak detection and specify the minimum distance between peaks. Then click "OK".

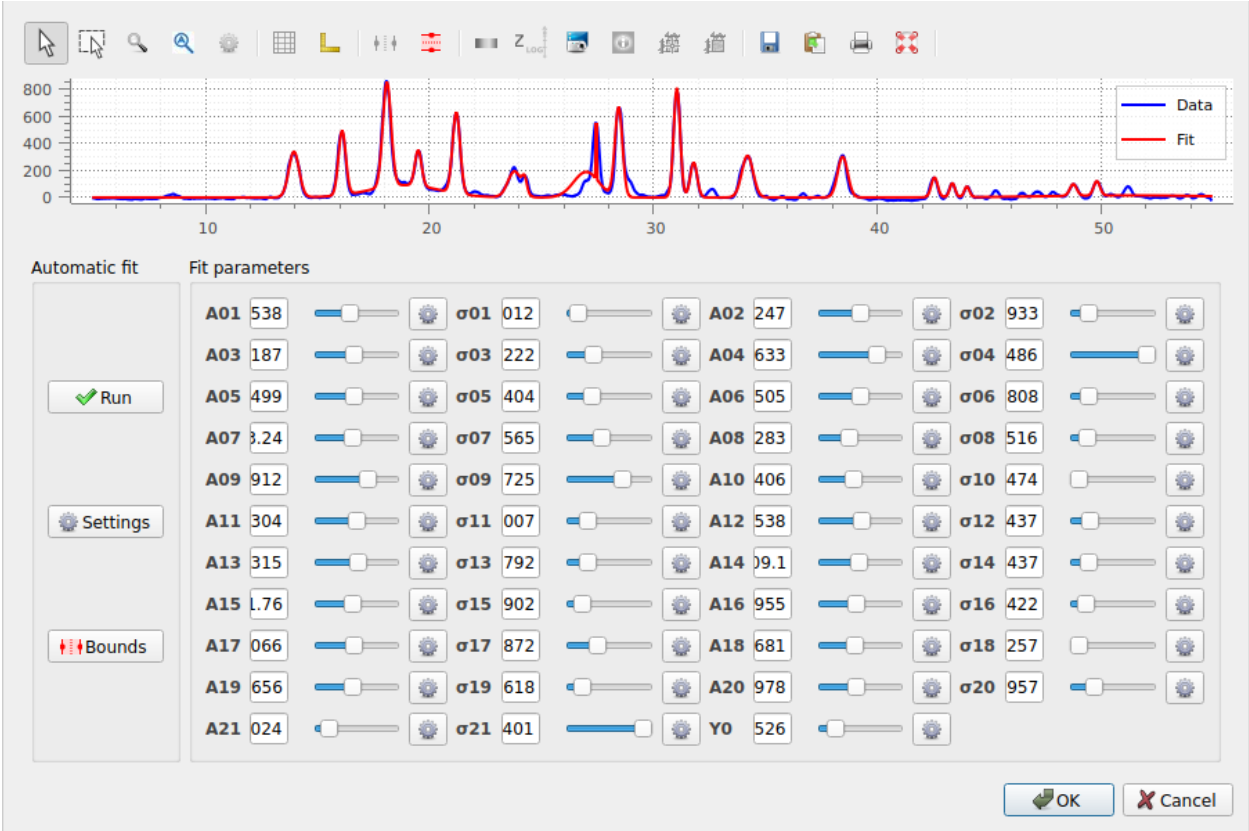


Fig. 27: The "Multi-Gaussian fit" dialog box appears. An automatic fit is performed by default. Click "OK" (or optionally adjust the parameters manually using the sliders, or modify the automatic fitting parameters).

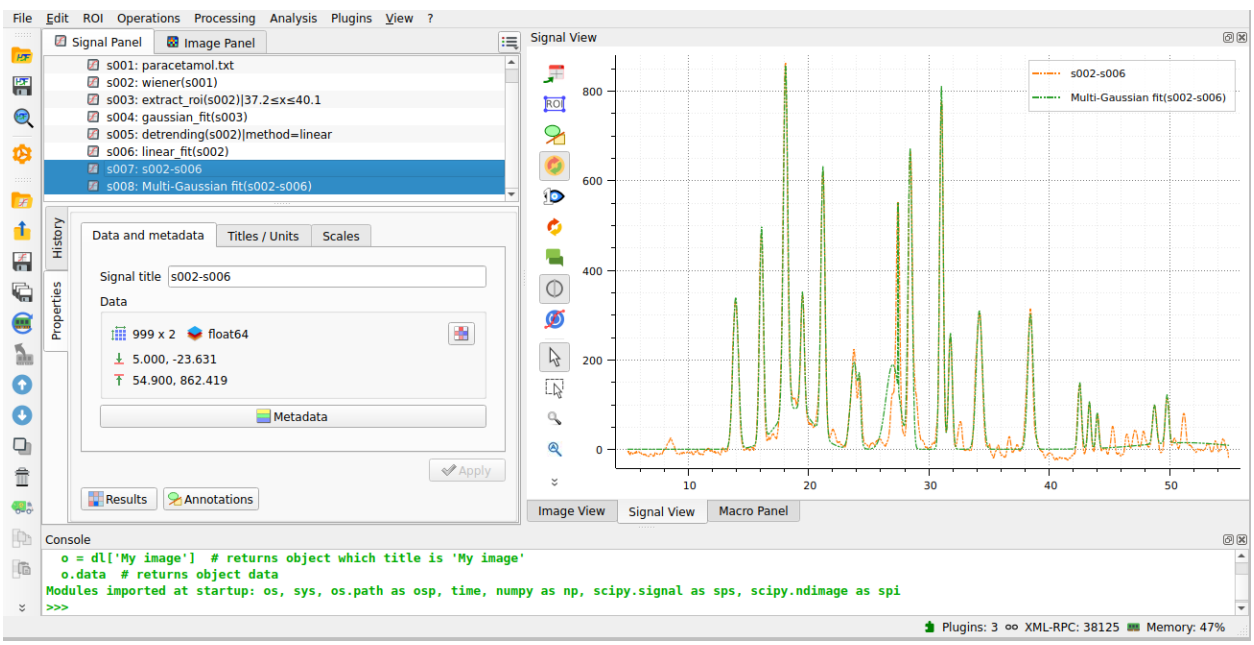


Fig. 28: The result of the fit is displayed in the main window. Here we selected both the spectrum and the fit in the "Signals" panel on the right, so both are displayed in the visualization panel on the left.

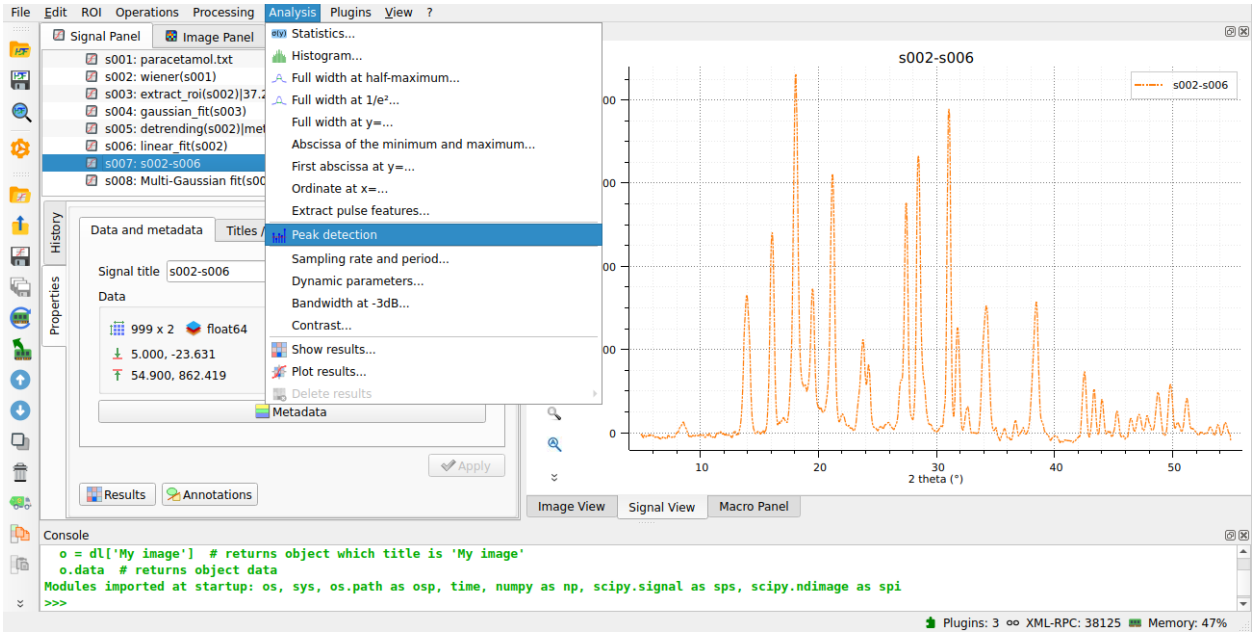


Fig. 29: Open the "Peak detection" window with "Analysis > Peak detection".

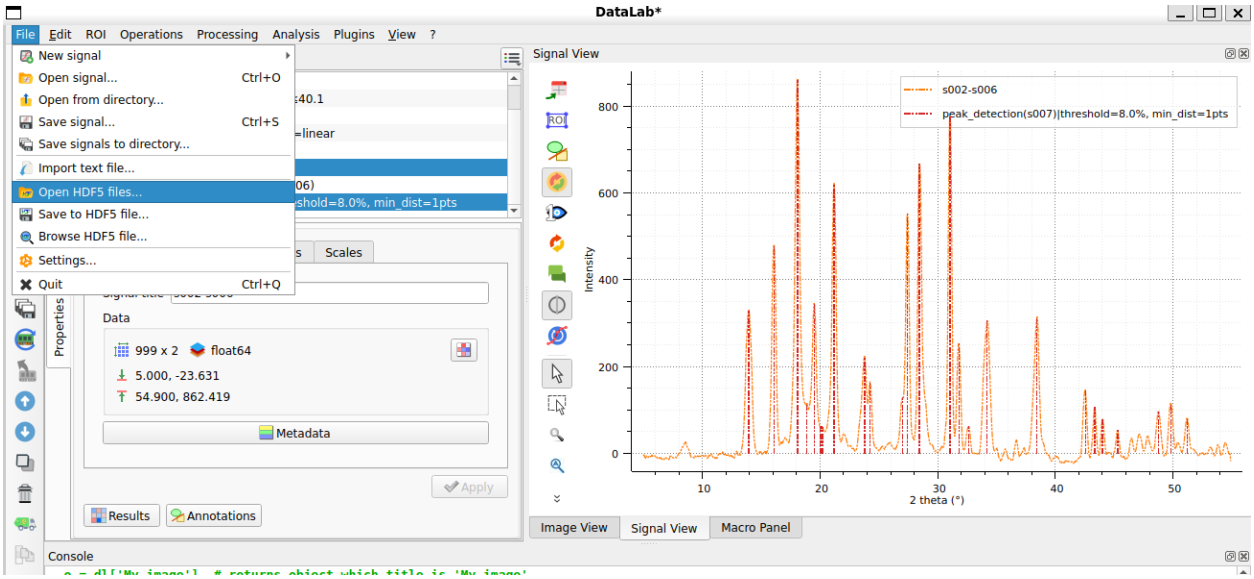


Fig. 30: After adjusting the peak detection parameters (using the same dialog as for the multi-Gaussian fit), click "OK". Then select both the "peak_detection" result and the original spectrum in the "Signals" panel to display them together in the visualization panel on the left.

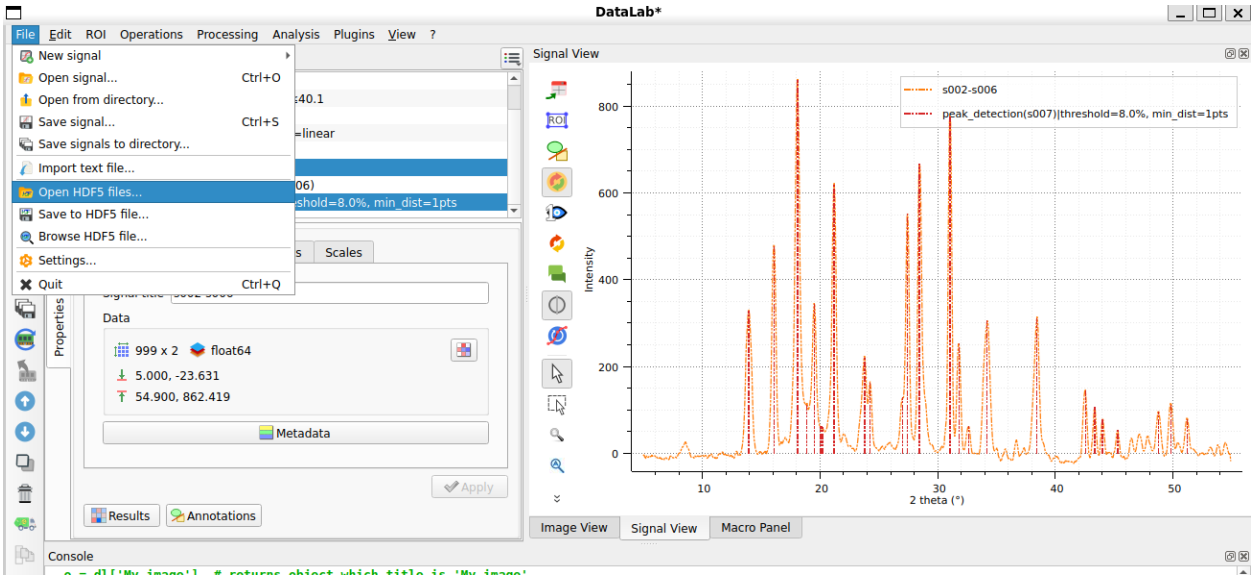


Fig. 31: Save the workspace to a file with "File > Save to HDF5 file…", or the button in the toolbar.

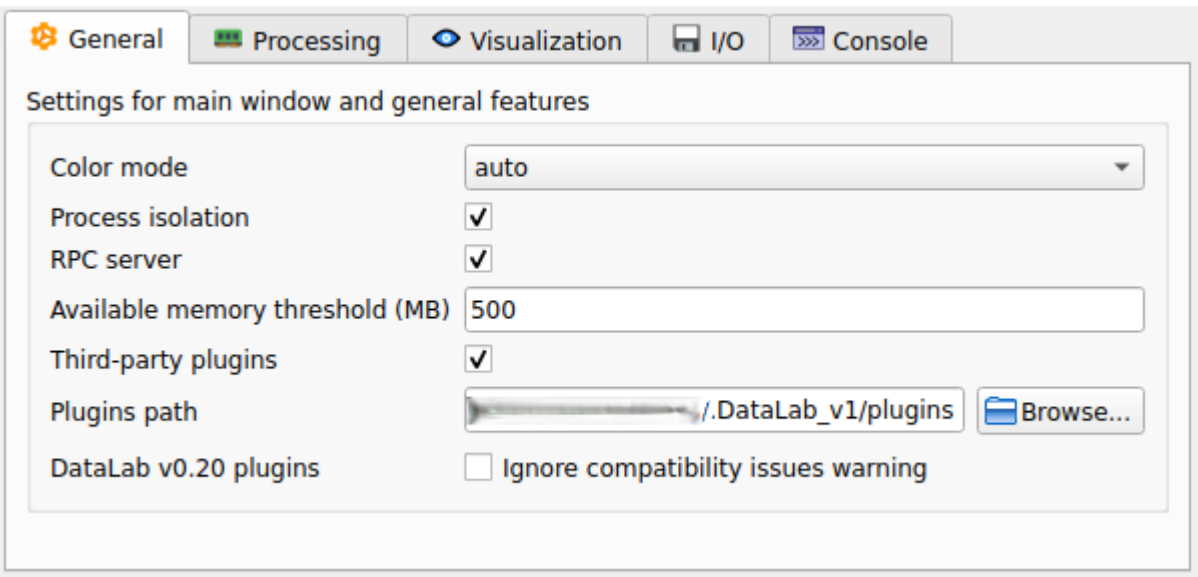


Fig. 32: In the "General" tab, we can see the "Plugins path" field. This is the path where DataLab will look for plugins. We can add a new plugin by copying/pasting the plugin file in this directory.

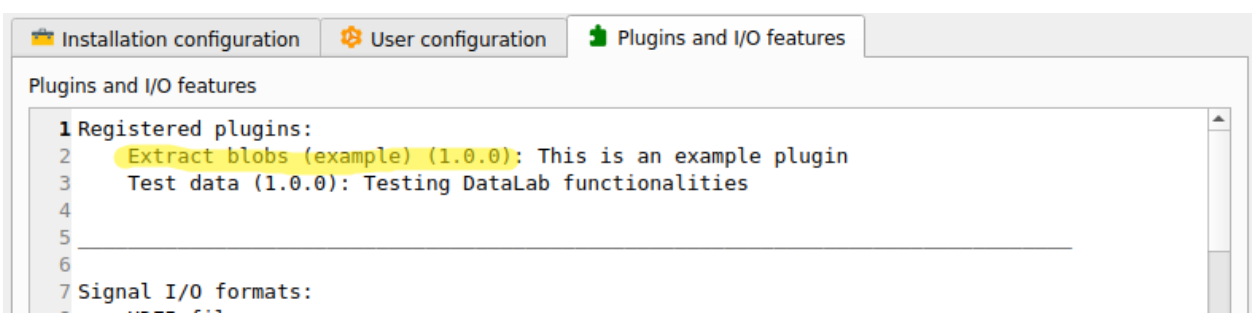


Fig. 33: The "About DataLab" dialog shows the list of available plugins.

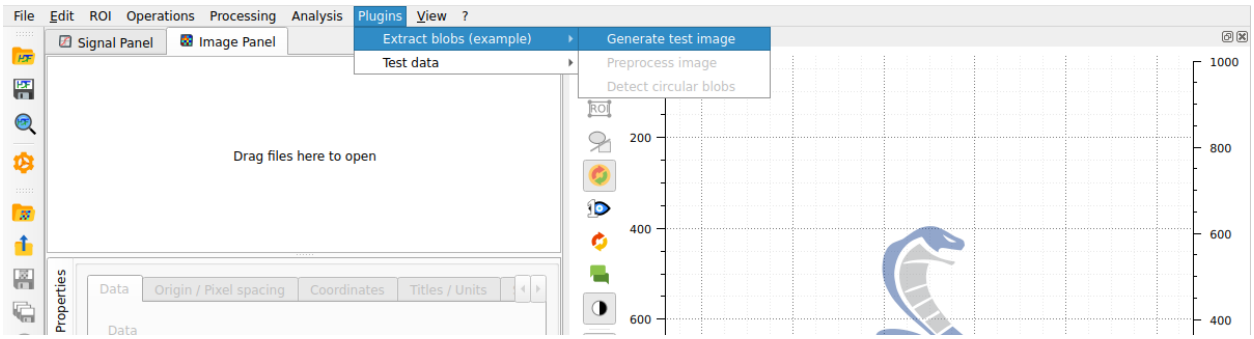


Fig. 34: Let's click on "Extract blobs (example) > Generate test image"

```
def generate_test_image(self) -> None:
    """Generate test image"""
    newparam = self.edit_new_image_parameters(
        title="Test image", hide_dtype=True, shape=(2048, 2048)
    )
    if newparam is not None:
        # Create a NumPy array:
        shape = (newparam.height, newparam.width)
        arr = np.random.normal(10000, 1000, shape)
        for _ in range(10):
            row = np.random.randint(0, shape[0])
            col = np.random.randint(0, shape[1])
            rr, cc = skimage.draw.disk((row, col), min(shape) // 50, shape=shape)
            arr[rr, cc] -= np.random.randint(5000, 6000)
        center = (shape[0] // 2,) * 2
        rr, cc = skimage.draw.disk(center, min(shape) // 10, shape=shape)
        arr[rr, cc] -= np.random.randint(5000, 8000)
        data = np.clip(arr, 0, 65535).astype(np.uint16)

        # Create a new image object and add it to the image panel
        obj = sigima.objects.create_image(
            newparam.title, data, units=("mm", "mm", "lsb")
        )
        self.proxy.add_object(obj)
```

The plugin has other features, such as denoising the image, and detecting blobs on the image, but we won't cover them here: we will instead use the DataLab native features to achieve, manually, the same operations as the plugin.

The image is a bit noisy, and also quite large. In addition, the blobs are large with respect to the pixel size.

To reduce the noise there are several functions available in DataLab. Due to the considerations we just made, we can consider that a binning would reduce the noise without losing the information we look for. Let's apply a binning to the image by a factor of 2 on both axes. This will reduce the image size by a factor of 4, and the noise standard deviation by a factor of 2. Choosing to change the pixel size accordingly will keep the blob size constant in the previous unit (in this case, pixels, but in a real application we can calibrate them to respect a real physical size).

A different approach we can take is to apply a moving median filter, to reduce the importance of the spikes. We do it with a window of 5; of course in practice different window sizes can be tested to find a good compromise between noise reduction and resolution. Let's see how to do that.

Now, we can be satisfied with our denoising: it is visible that the noise is smaller. We can thus move to the following

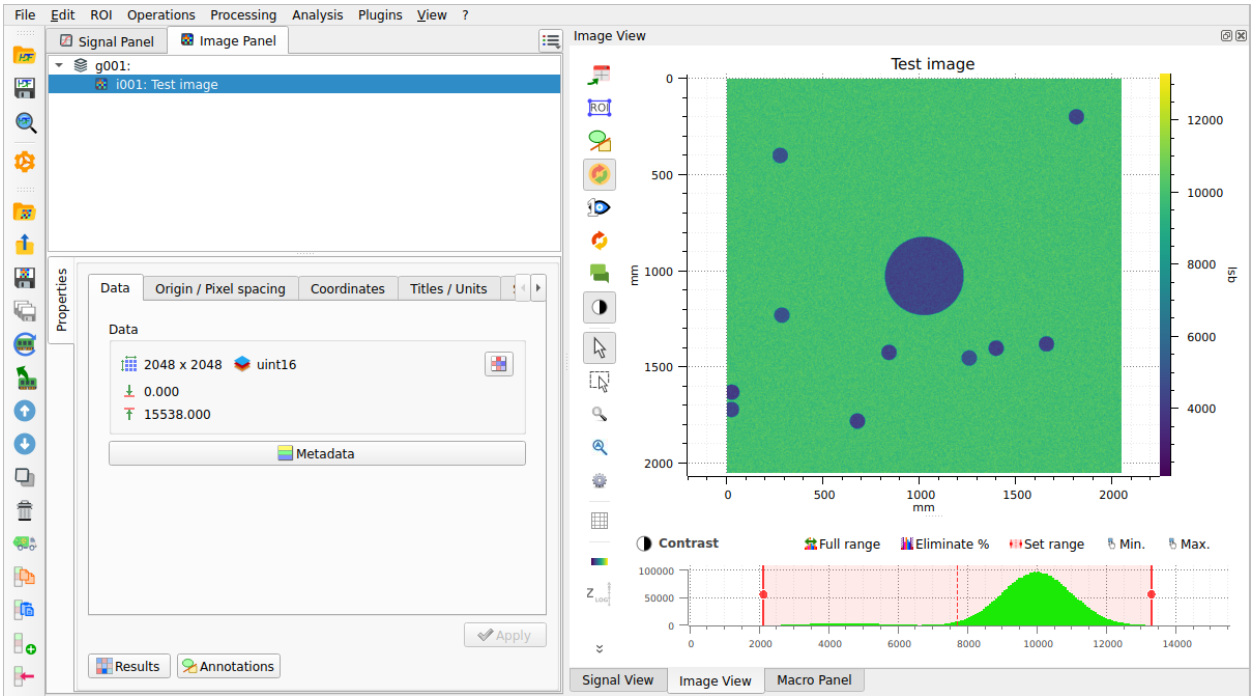


Fig. 35: The plugin has generated a test image, and added it to the "Images" panel. The image shows a few blobs, with a central bigger disk, and a noisy background.

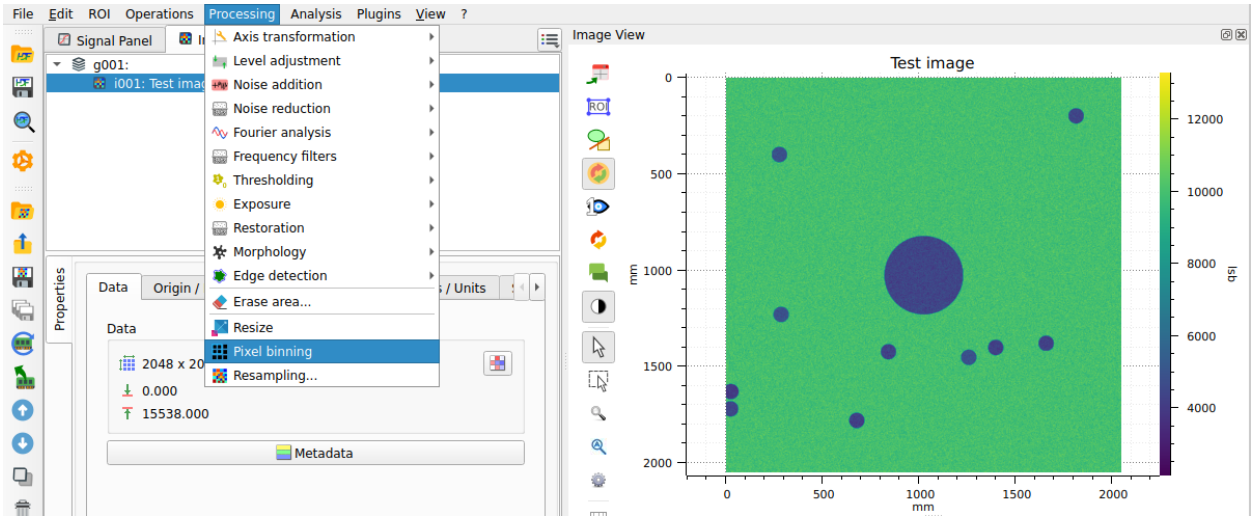


Fig. 36: Click on "Operations > Pixel binning".

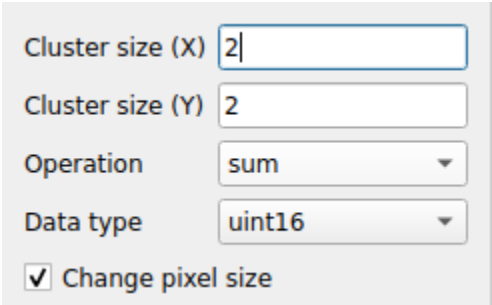


Fig. 37: The "Binning" dialog opens, with several options. Set the binning factor to 2 for both axes, select the "change pixel size" case and click on "OK".

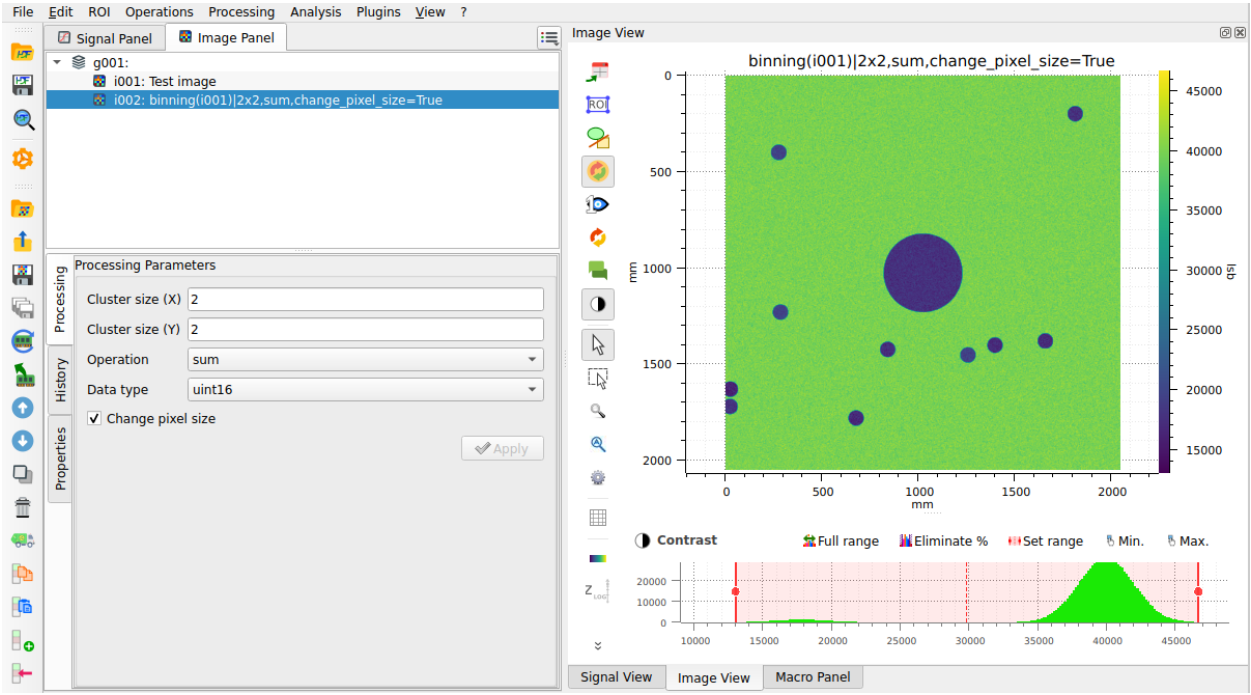


Fig. 38: The binned image is added to the "Images" panel. It is now easier to see the blobs (even if they were already quite visible on the original image: this is just an example), and the image will be faster to process.

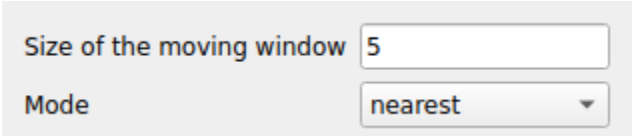


Fig. 39: Click on "Processing > Noise reduction > Moving median" entry, and set the window size to 5.

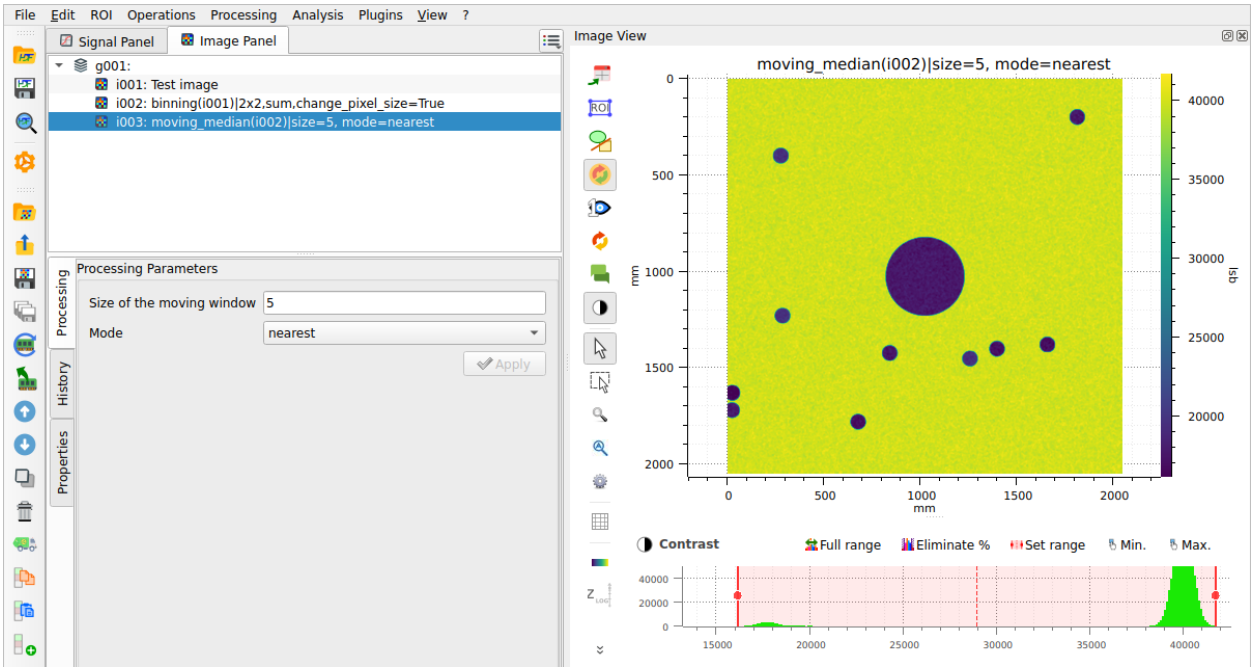


Fig. 40: The filtered image is added to the "Images" panel. Denoising is quite efficient.

step: detect the blobs on the image. To do that, we can use the "Blob detection" function available in the "Analysis" menu. Different *algorithms* are available, here we will use the OpenCV one.

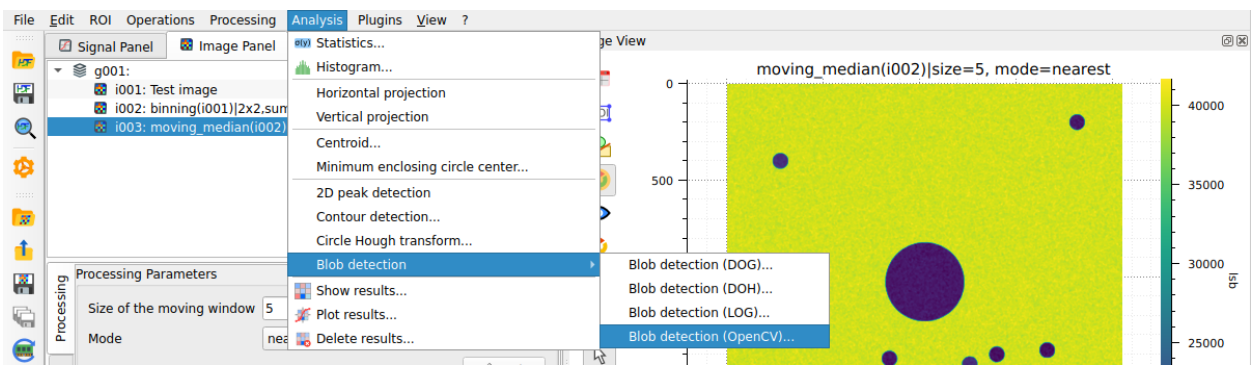


Fig. 41: Click on "Analysis > Blob detection > Blob detection (OpenCV)".

Note: If you want to show the analysis results again, you can select the "Show results" entry in the "Analysis" menu, or the "Results" button, in the property panel below the image list:

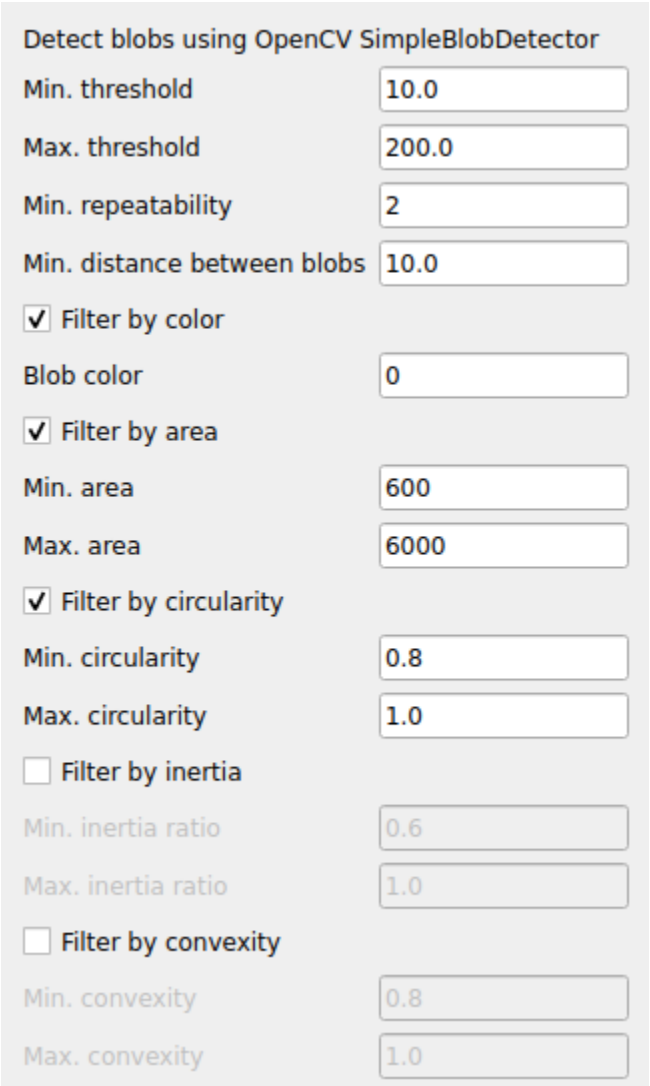


Fig. 42: The "Blob detection (OpenCV)" dialog opens. Set the parameters as shown on the screenshot, and click on "OK".

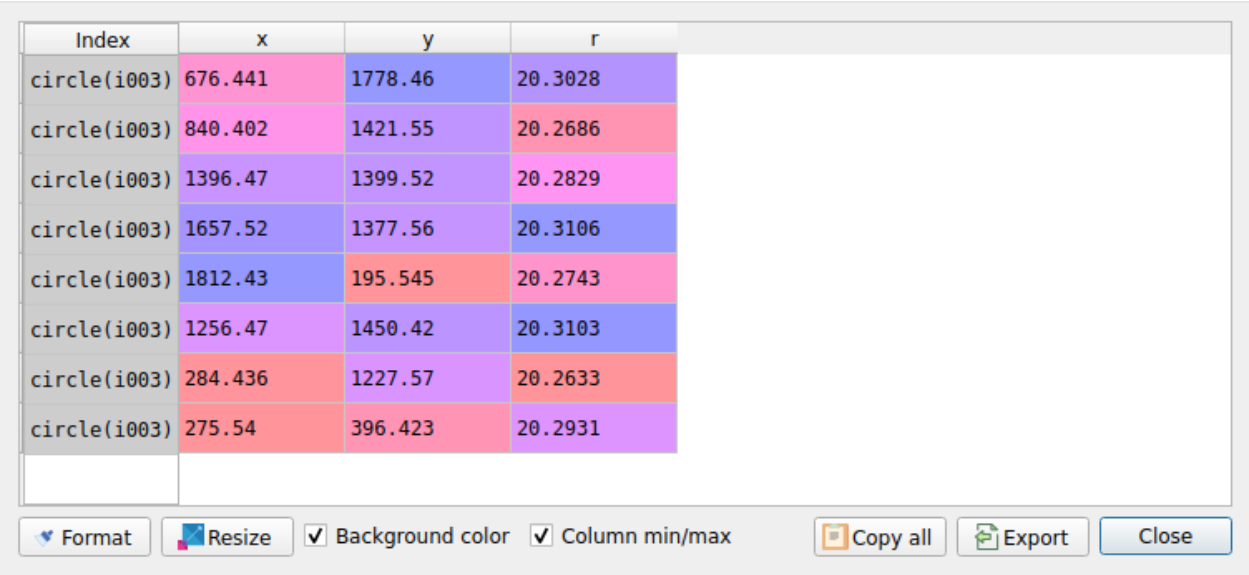


Fig. 43: The "Results" dialog opens, showing the detected blobs: one line per blob, with the blob coordinates and radius.

Finally, we can save the workspace to a file. The workspace contains all the images that were loaded in DataLab, as well as the processing results. It also contains the visualization settings (colormaps, contrast, etc.), the metadata, and the annotations. To save the workspace, click on "File > Save to HDF5 file…", or the button in the toolbar.

If you want to load the workspace again, you can use the "File > Open HDF5 file…" (or the button in the toolbar) to load the whole workspace, or the "File > Browse HDF5 file…" (or the button in the toolbar) to load only a selection of data sets from the workspace.

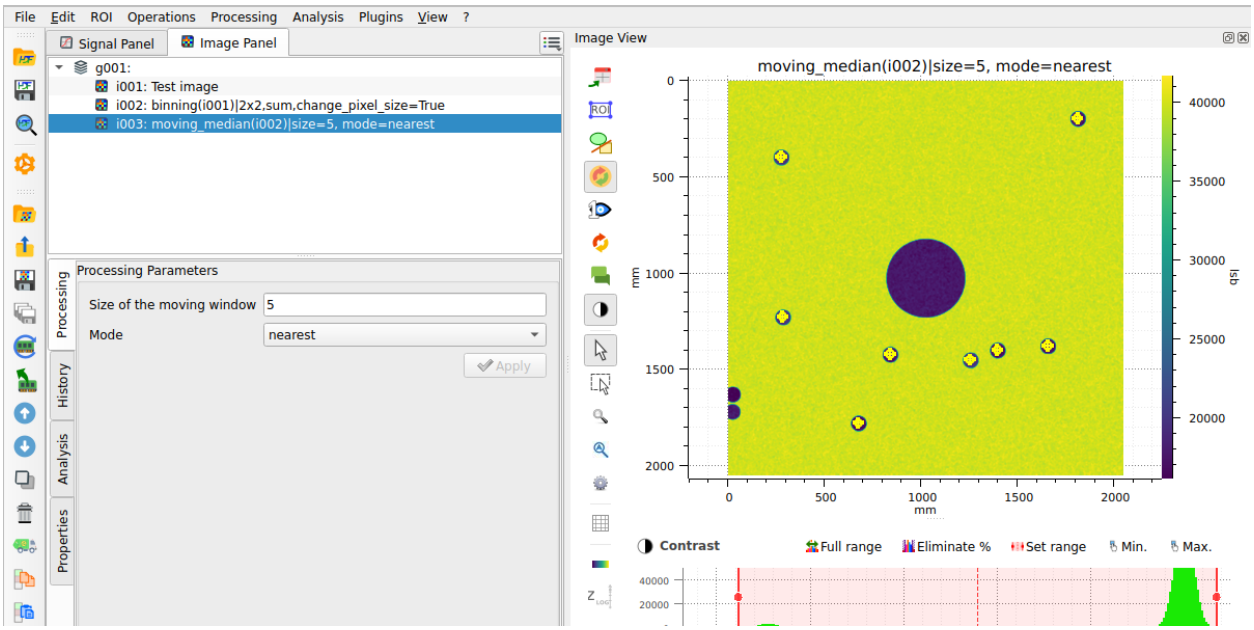


Fig. 44: The detected blobs are also added to the image metadata, and can be seen in the visualization panel on the right.

Measuring Fabry-Perot fringes

This example shows how to measure Fabry-Perot fringes using the image processing features of DataLab:

- Load an image of a Fabry-Perot interferometer
- Define a circular region of interest (ROI) around the central fringe
- Detect contours in the ROI and fit them to circles
- Show the radius of the circles
- Annotate the image
- Copy/paste the ROI to another image
- Extract the intensity profile along the X axis
- Save the workspace

Load the images

Note: The images used in this tutorial "fabry_perot1.jpg" and "fabry_perot2.jpg" are available in the tutorial data folder of DataLab's installation directory (`<DataLab installation directory>/data/tutorials/`).

Alternatively, they can be downloaded from the online documentation at https://datalab-platform.com.

First, we open DataLab and load the images from the menu file "File > Open Image…", with the button in the toolbar or by dragging and dropping the files into DataLab.

The selected image is displayed in the main window. We can zoom in and out by pressing the right mouse button and dragging the mouse up and down. We can also pan the image by pressing the middle mouse button and dragging the

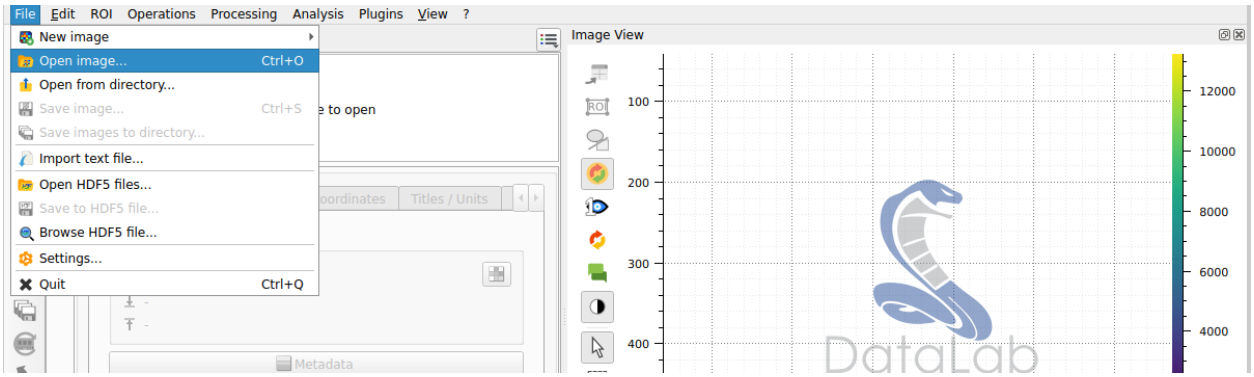


Fig. 45: Open the image files with "File > Open Image…", or with the button in the toolbar, or by dragging and dropping the files into DataLab (on the panel on the right).

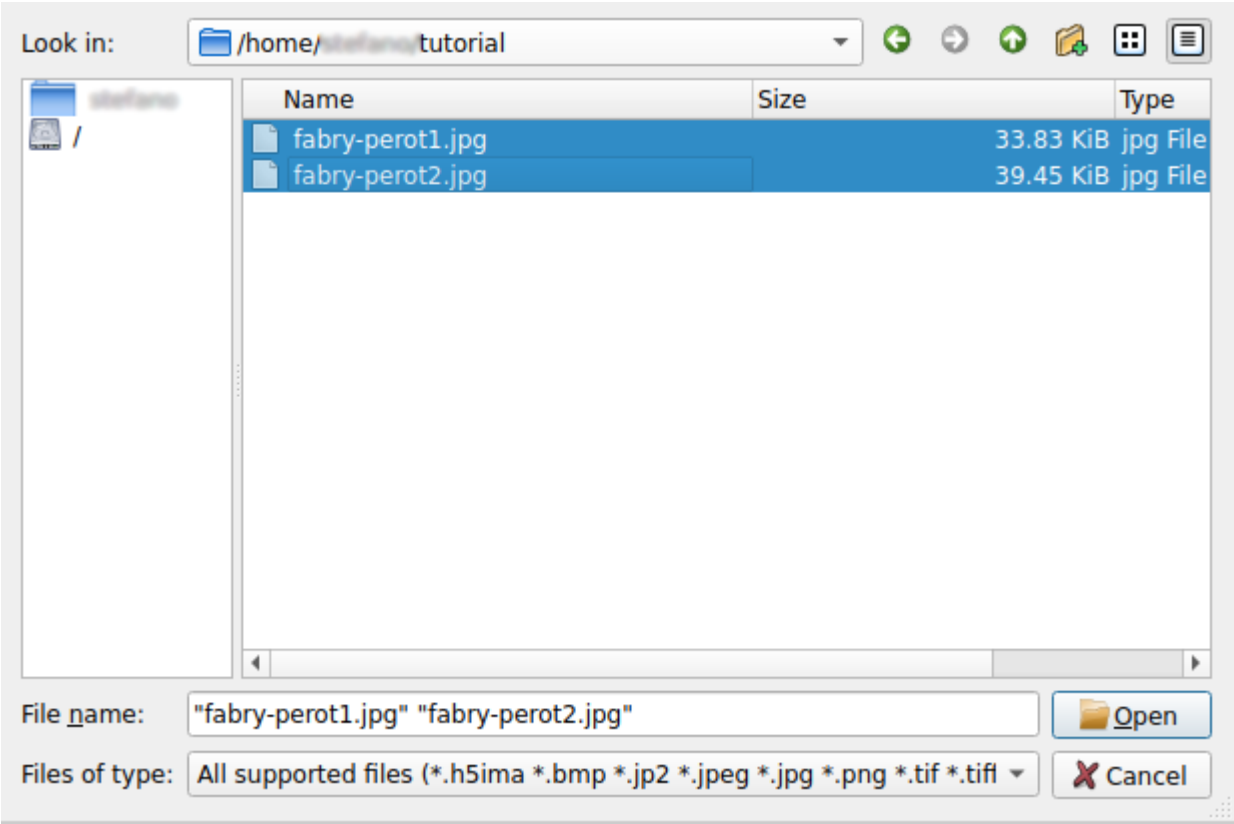


Fig. 46: Go to the tutorial data folder in DataLab's installation directory (or the folder where you placed the images), select "fabry_perot1.jpg" and "fabry_perot2.jpg" and click "Open".

mouse.

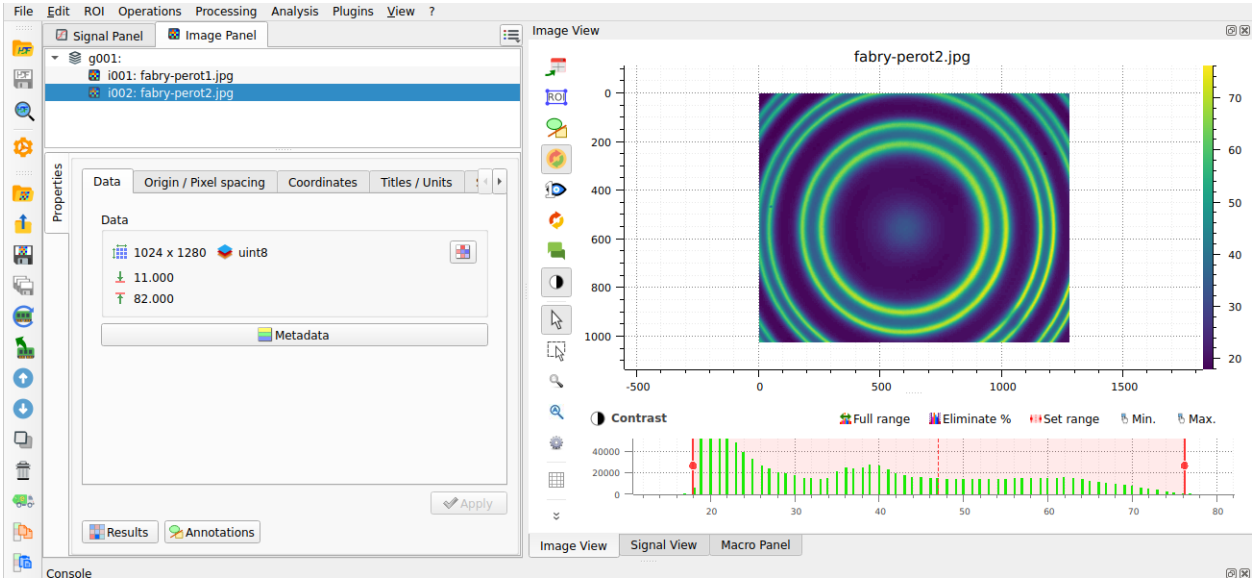


Fig. 47: Images loaded in DataLab. Zoom in and out with the right mouse button, pan the image with the middle mouse button.

Note: When working on application-specific images (e.g. X-ray radiography images, or optical microscopy images), it is often useful to change the colormap to a grayscale colormap. If you see a different image colormap than the one shown in the figure, you can change it by selecting the image in the visualization panel, and the selecting the colormap in the vertical toolbar on the left of the visualization panel.

Or, even better, you can change the default colormap in the DataLab settings by selecting "Edit > Settings…" in the menu, or the button in the toolbar.

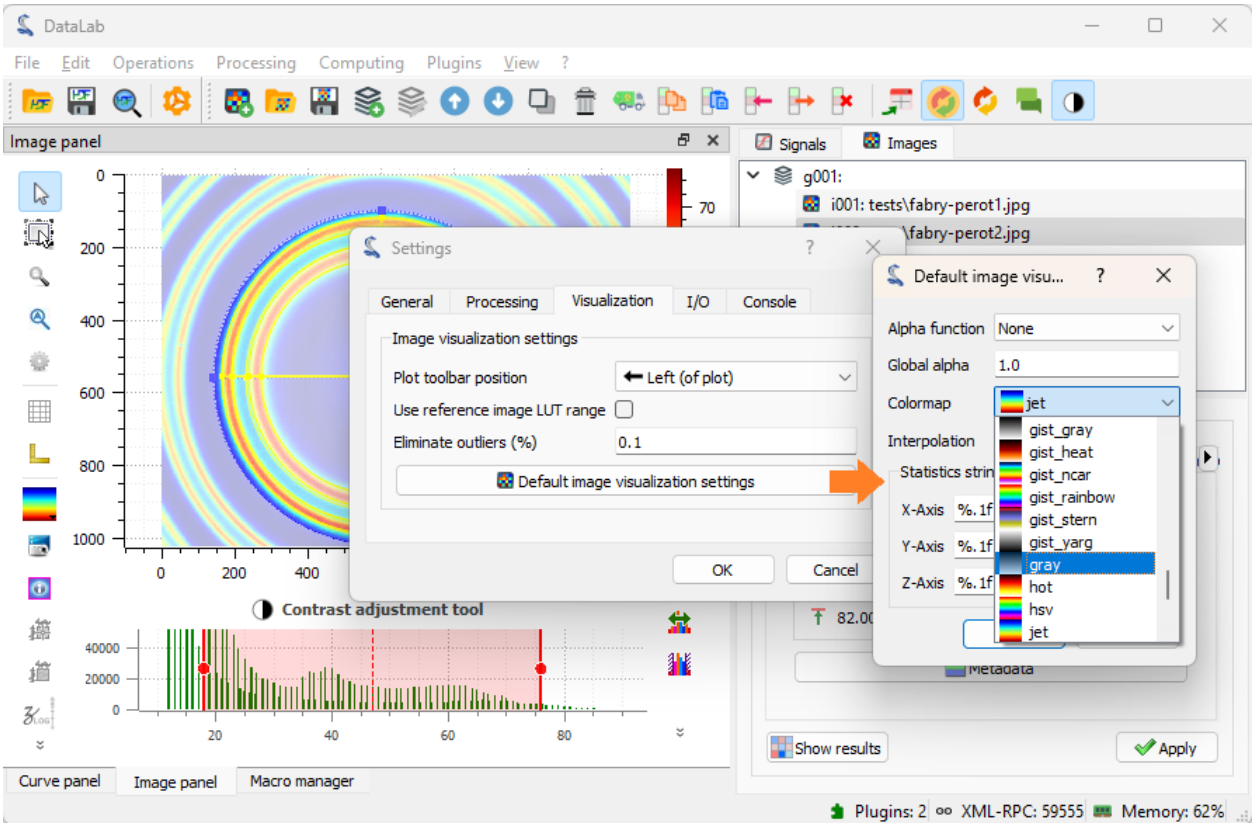


Fig. 48: Select the "Visualization" tab, and select the "gray" colormap.

Define circular ROIs and fit contours

Let's define a circular region of interest (ROI) around the central fringe. To do that, we firstly select the first image in the "Images" panel (if not already selected), then we select the "Edit graphically" tool in the "ROI" menu.

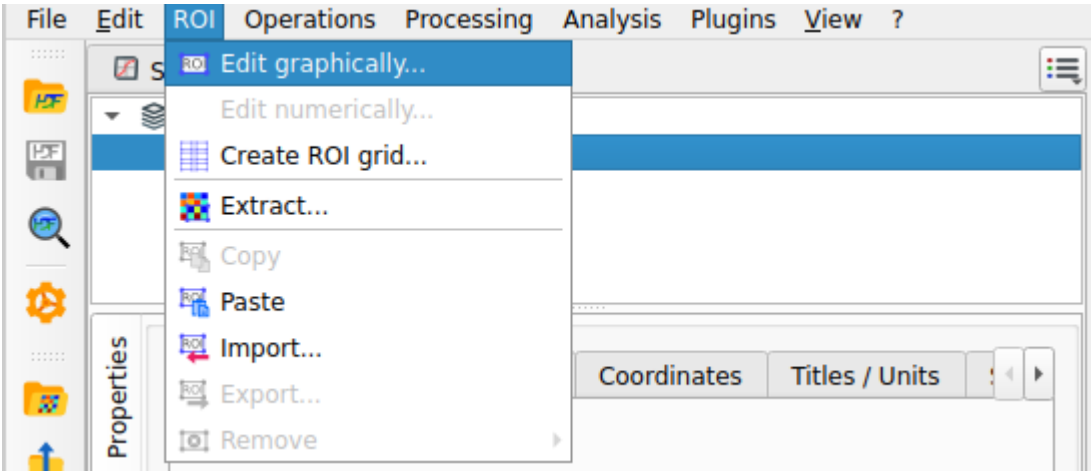


Fig. 49: Select the "Edit graphically" tool in the "ROI" menu.

This opens the "Regions of interest" dialog, that provides all the tools to define and edit ROIs. You can define ROIs graphically or via their coordinates. You can define multiple ROIs on the same image, and even combine them using boolean operations (union, intersection, difference).

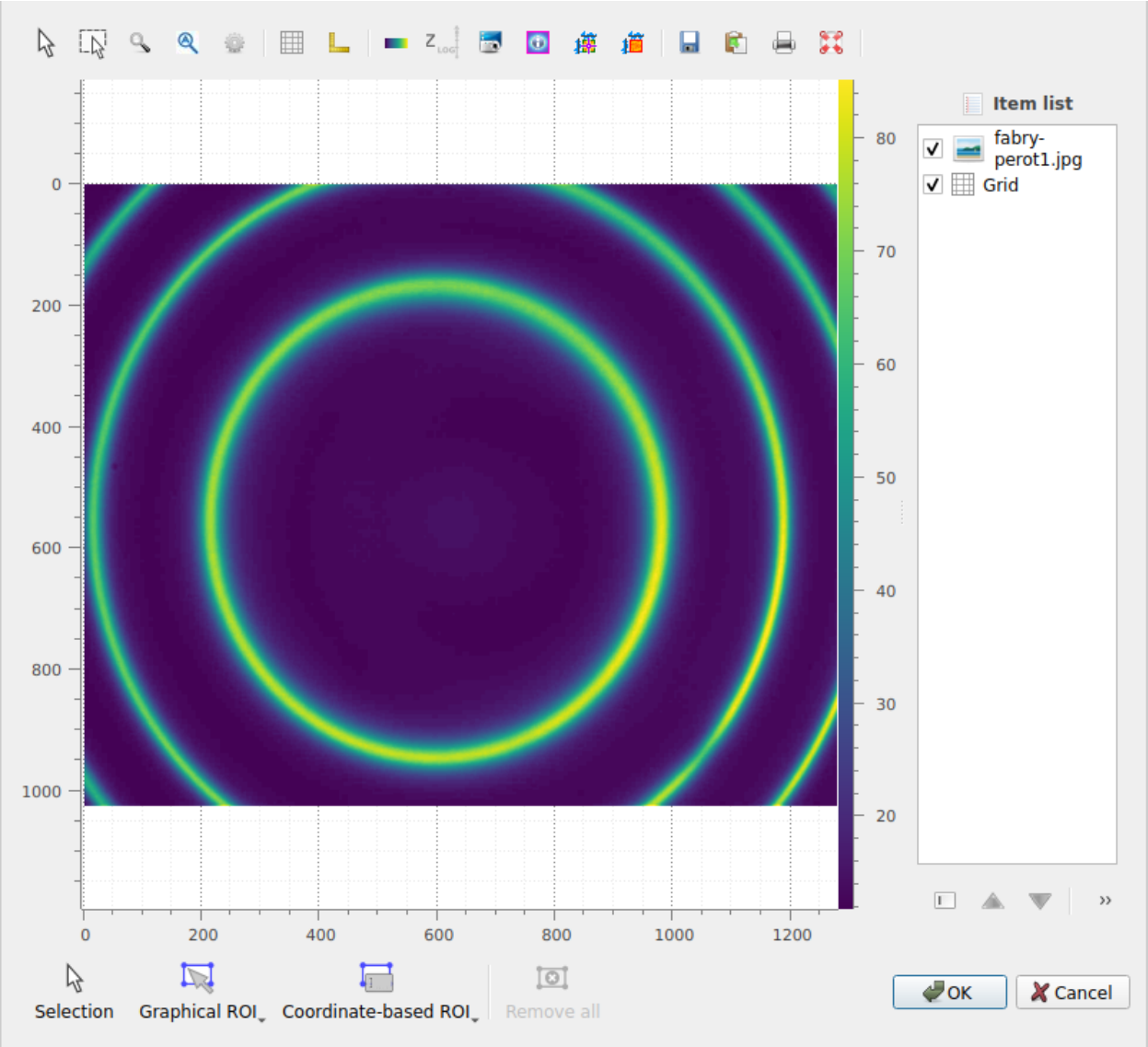


Fig. 50: The "Regions of interest" dialog opens. Click "Add ROI" and select a circular ROI. Resize the predefined ROI by dragging the handles. Note that you may change the ROI radius while keeping its center fixed by pressing the "Ctrl" key. Click "OK" to close the dialog.

We choose to define a circular ROI graphically. To do that, we click on the "Graphical ROI" button, and select the circular ROI option. We can now draw the circular ROI on the image by clicking and dragging the mouse: the first click is on one side of the circle, and the second click is on the opposite side of the circle.

Once created, the ROI can be selected using the "Selection" tool, and moved or resized by dragging the ROI or its handles. In addition, you can change the ROI radius while keeping its center fixed by pressing the "Ctrl" key while dragging a handle.

Once you are satisfied with the ROI position and size, click "OK" to close the dialog. A confirmation dialog opens to ask you to apply the ROI to the image, and confirm its parameters. Here you can also choose to invert the ROI mask (i.e., select the area outside the ROI instead of the area inside the ROI): we do not want to invert the ROI in this case,

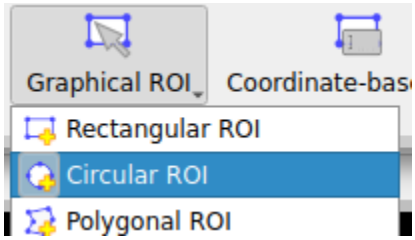


Fig. 51: The option to graphically define a circular ROI.

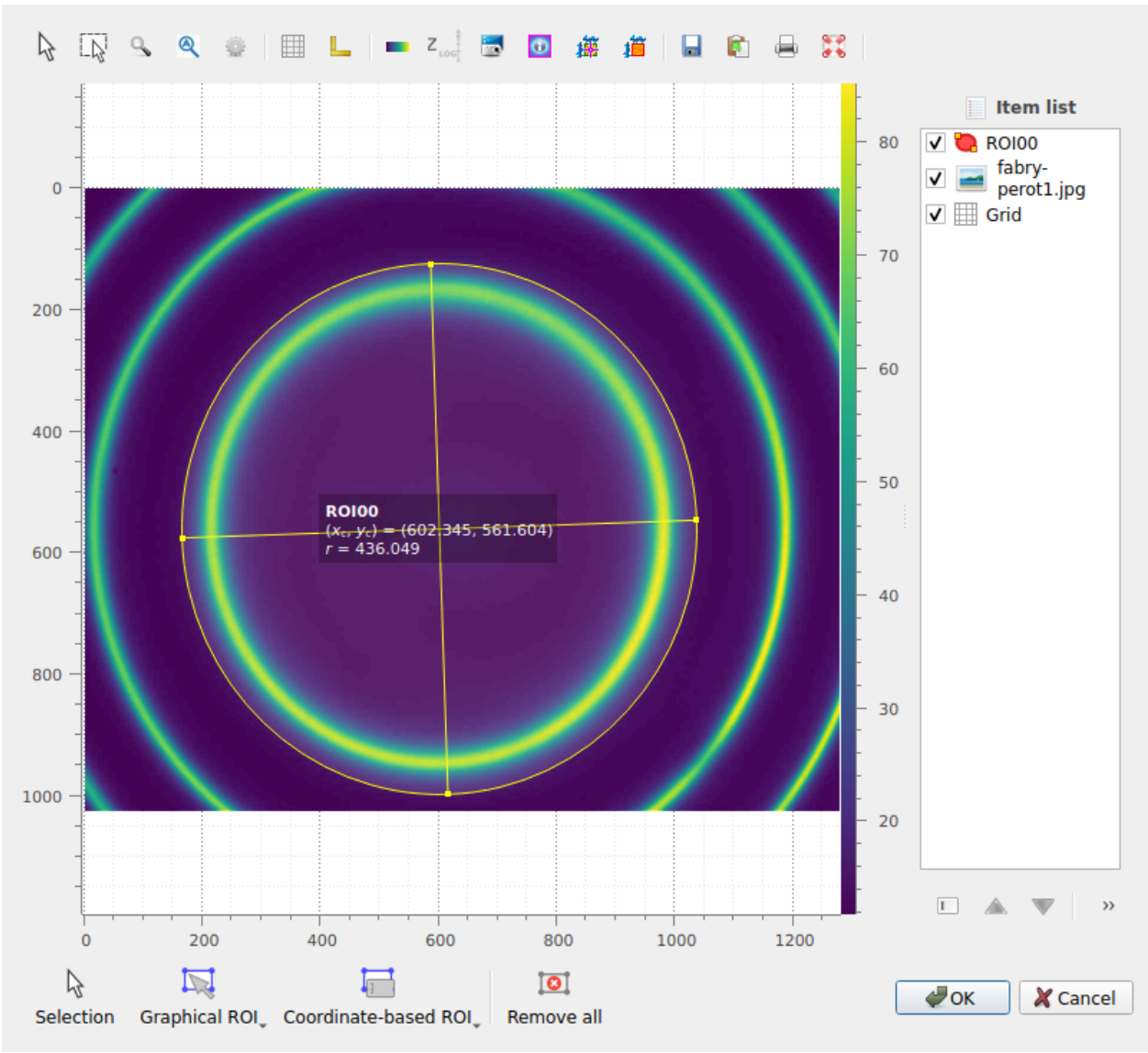


Fig. 52: The circular ROI created appears in the image.

so we leave the "Invert ROI" checkbox unchecked, and click "OK". If you want to obtain the exact results shown in this tutorial, make sure that the ROI parameters are the same as in the figure.

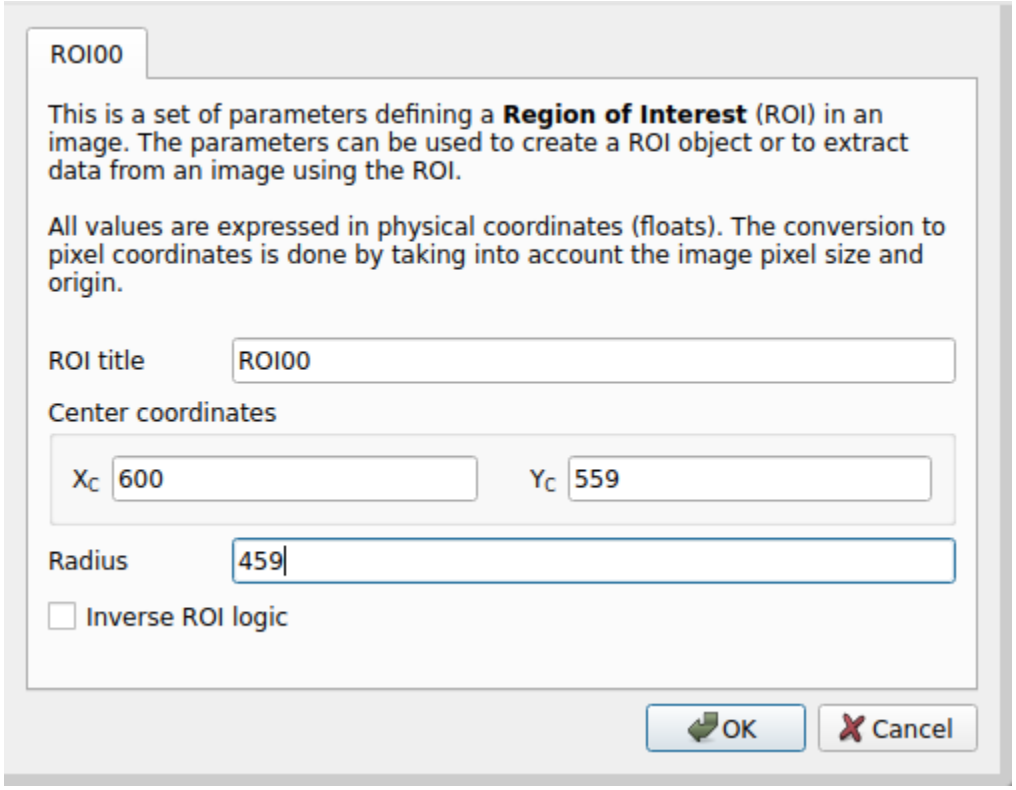


Fig. 53: The confirmation dialog.

Once confirmed, the ROI is applied to the image. The ROI is displayed on the image, and the part outside the ROI is dimmed. Note that, internally, the ROI is defined by a binary mask, i.e., image data is represented as a NumPy masked array.

Now, let's detect the contours in the area inside the ROI and fit them to circles. To do that, we select the "Contour detection" tool in the "Analysis" menu. The contour parameters dialog opens: we select the shape "Circle" and click "OK". Now you will see a "Results" dialog opening, that displays the fitted circle parameters. Click "OK" to close the dialog. The fitted circles are displayed on the image.

Note: If you want to show the analysis results again, you can select the "Show Results" button in the "Analysis" menu, or the "Results" button, in the Analysis panel located below the image list:

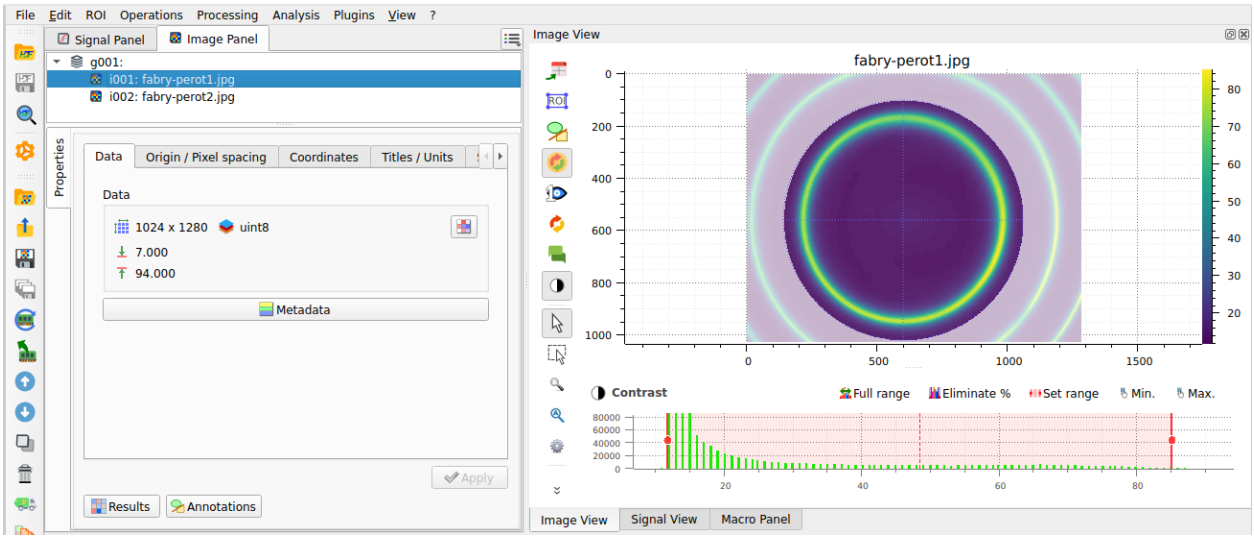


Fig. 54: The ROI displayed on the image.

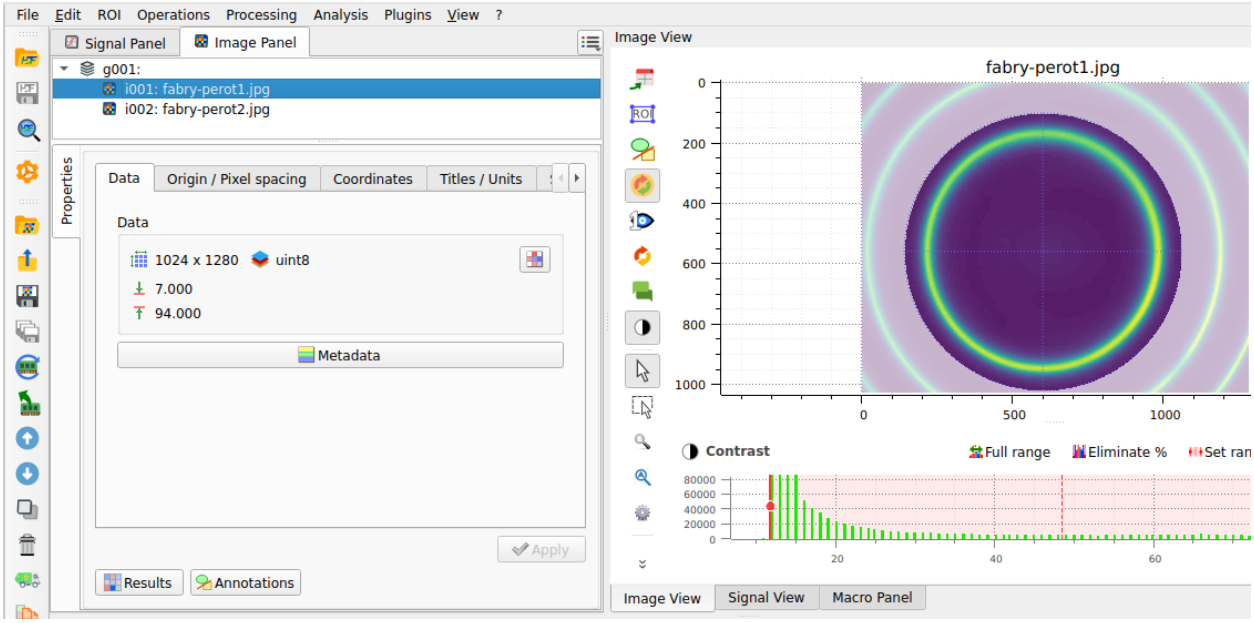


Fig. 55: The “Contour detection” tool in the “Analysis” menu.

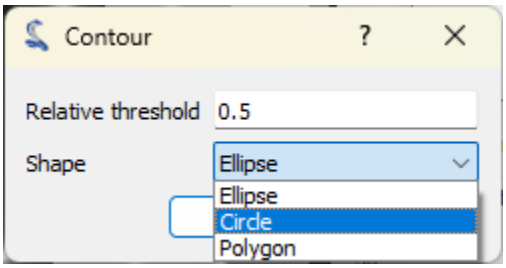


Fig. 56: The “Contour” parameters dialog.

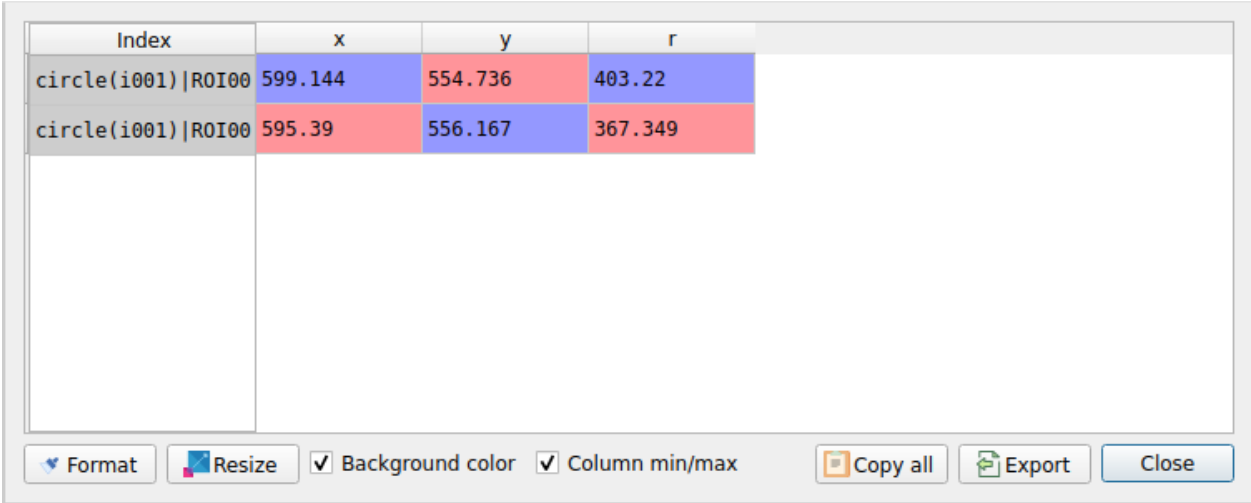


Fig. 57: The "Results" dialog.

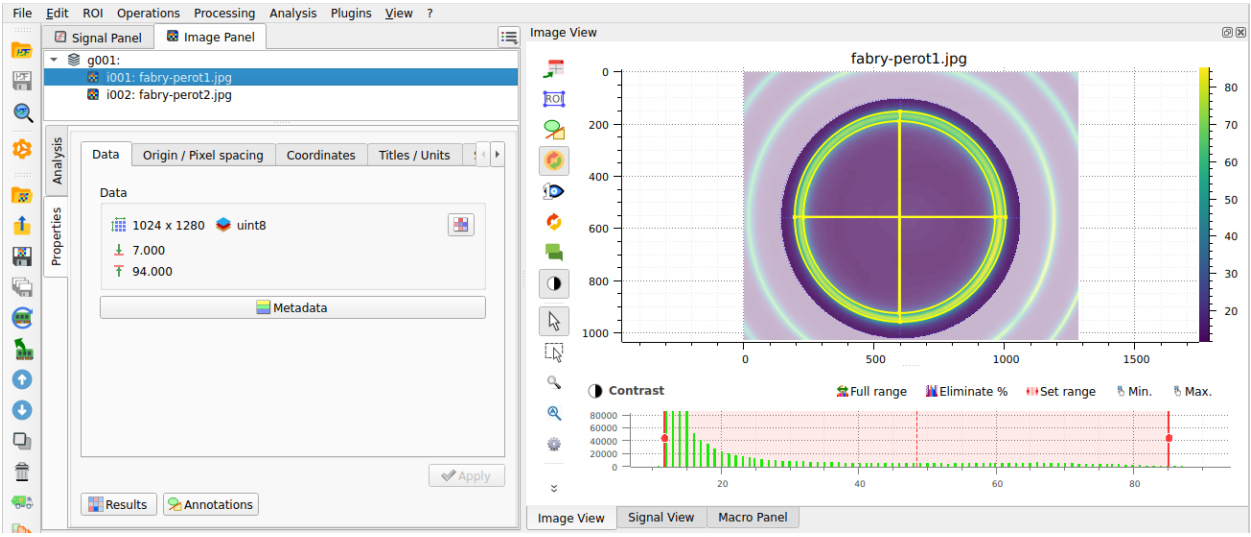


Fig. 58: The fitted circles are displayed on the image.

The images (or signals) can also be displayed in a separate window, by clicking on the "View in a new window" entry in the "View" menu (or the button in the toolbar). This is useful to compare images or signals side by side, and allows you to add annotations (cursors, labels, shapes, etc.) to the image or signal: the annotations are stored in the metadata of the image or signal, and saved together with the data when the workspace is saved.

If you want to take a closer look at the metadata, you can open the "Metadata" dialog.

Now, let's delete the image metadata (including the annotations) to clean up the image. Select the "Delete object metadata…" entry in the "Edit" menu, or the button in the toolbar, and answer "No" to the confirmation dialog to keep the ROI and delete the rest of the metadata.

If we want to define the exact same ROI on the second image, we can copy/paste the ROI from the first image to the second image. The best way to do that is to use the copy/paste option present in the "ROI" menu, or the and buttons in the toolbar: select the first image, then click on the "Copy" entry in the "ROI" menu (or the button in the toolbar), then select the second image, and click on the "Paste ROI" entry in the "ROI" menu (or the button in the toolbar).

Note: Copying/pasting the metadata copies/pastes all the metadata, including the ROIs, but this would have the side effect of also copying/pasting the rest of the metadata.

Select the "Contour detection" tool in the "Analysis" menu, with the same parameters as before (shape "Circle"). On this image, there are two fringes, so four circles are fitted. The "Results" dialog opens and displays the fitted circle parameters. Click "OK".

Extract intensity profiles along the X axis

Note: For the sake of simplicity (and because we want to compare two methods of extraction), we will extract the intensity profile along the X axis at the center of the image. In a real application, you would probably want to extract the **radial intensity profile** instead, which can be done using the "Radial profile" entry in the "Analysis > Intensity profiles" menu: this would be more relevant and straightforward to analyze Fabry-Perot fringes.

To extract the intensity profile along the X axis, we have two options:

- Either select the "Line profile…" entry in the "Analysis > Intensity profiles" menu.
- Or activate the "Cross section" tool in the vertical toolbar on the left of the visualization panel.

Let's try the first option: select the "Line profile…" entry . This is the most straightforward way to extract a profile from an image, and it corresponds to the `calc("line_profile")` computation feature of DataLab's API (so it can be used in a script, a plugin or a macro). Select the "Line profile…" entry in the "Operations" menu and the "Line

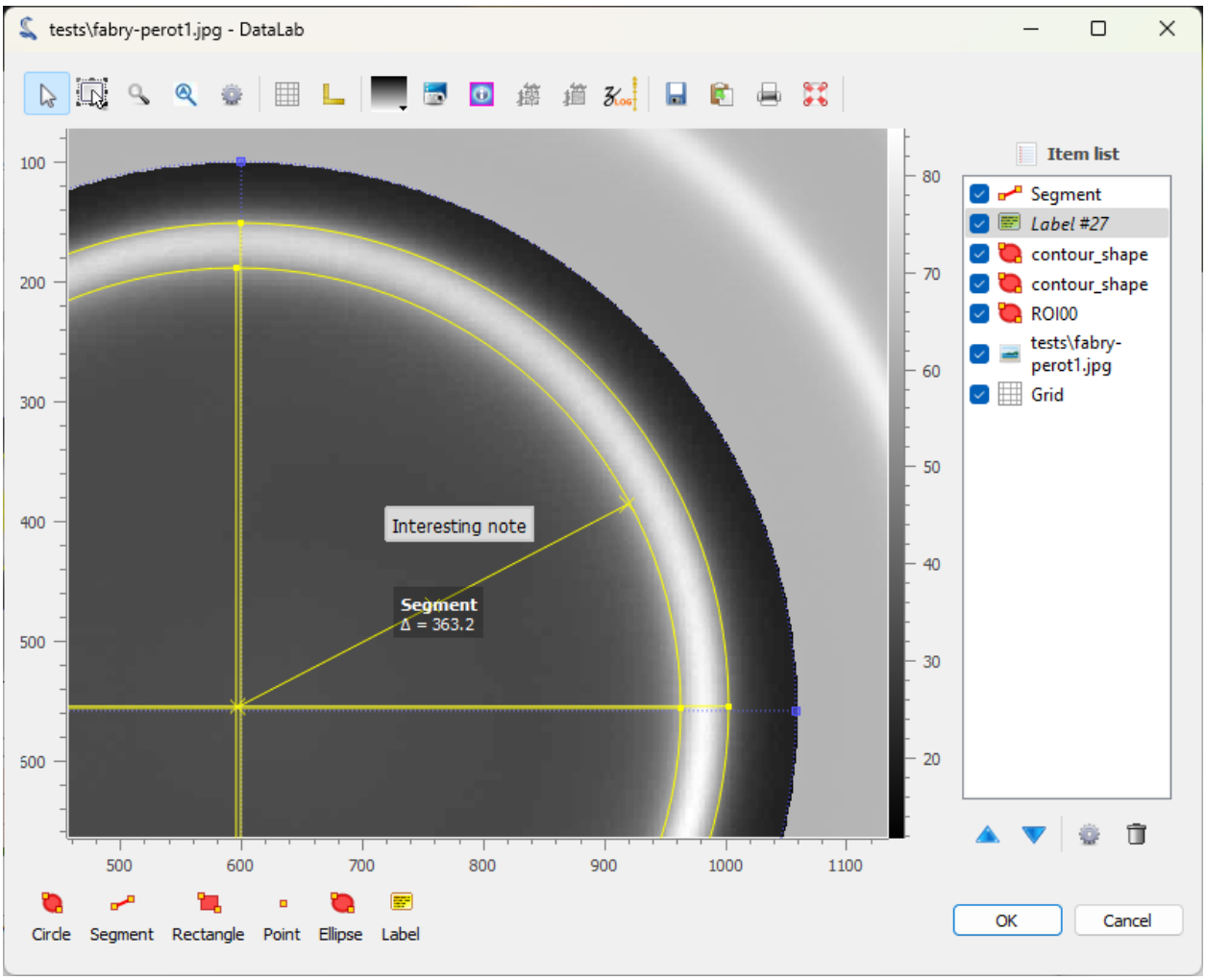


Fig. 59: The image displayed in a separate window. The ROI and the fitted circles are also displayed. Annotations can be added to the image by clicking on the buttons at the bottom of the window. The annotations are stored in the metadata of the image, and saved together with the image data when the workspace is saved. Click on "OK" to close the window.

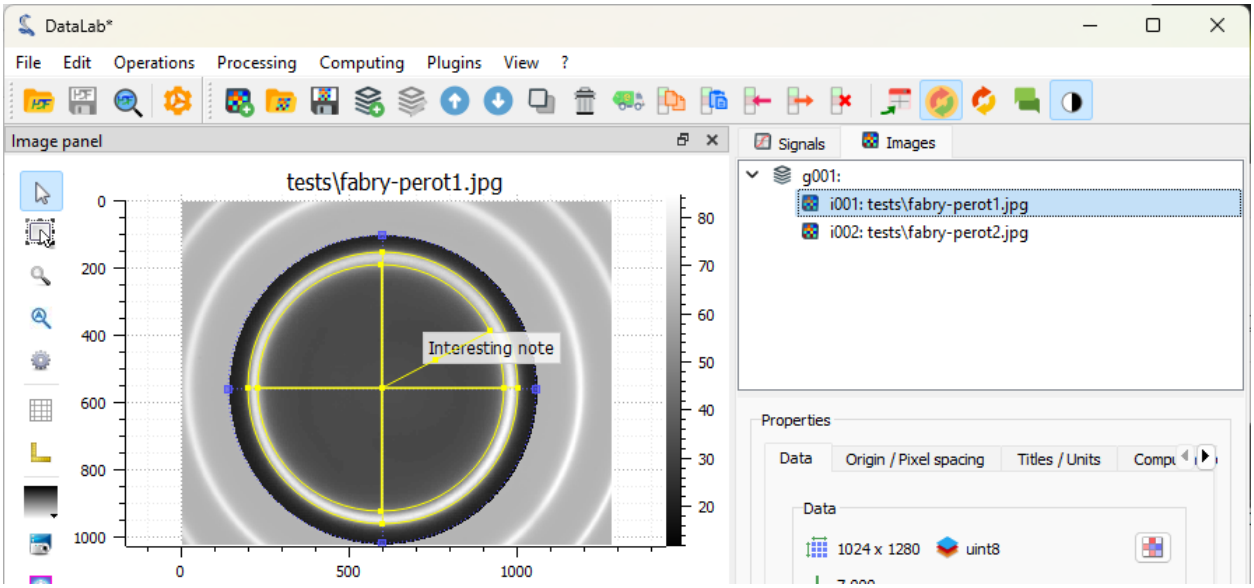


Fig. 60: The image is displayed in the main window, together with the annotations.

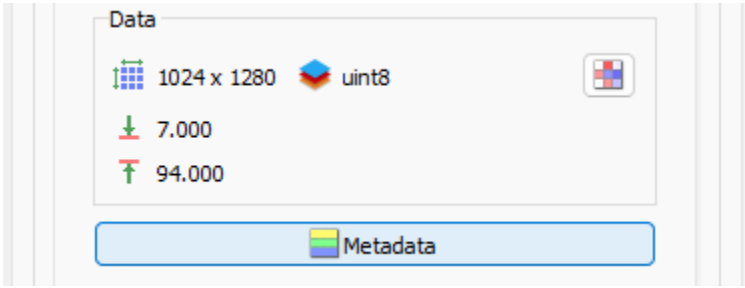


Fig. 61: The "Metadata" button is located below the image list.

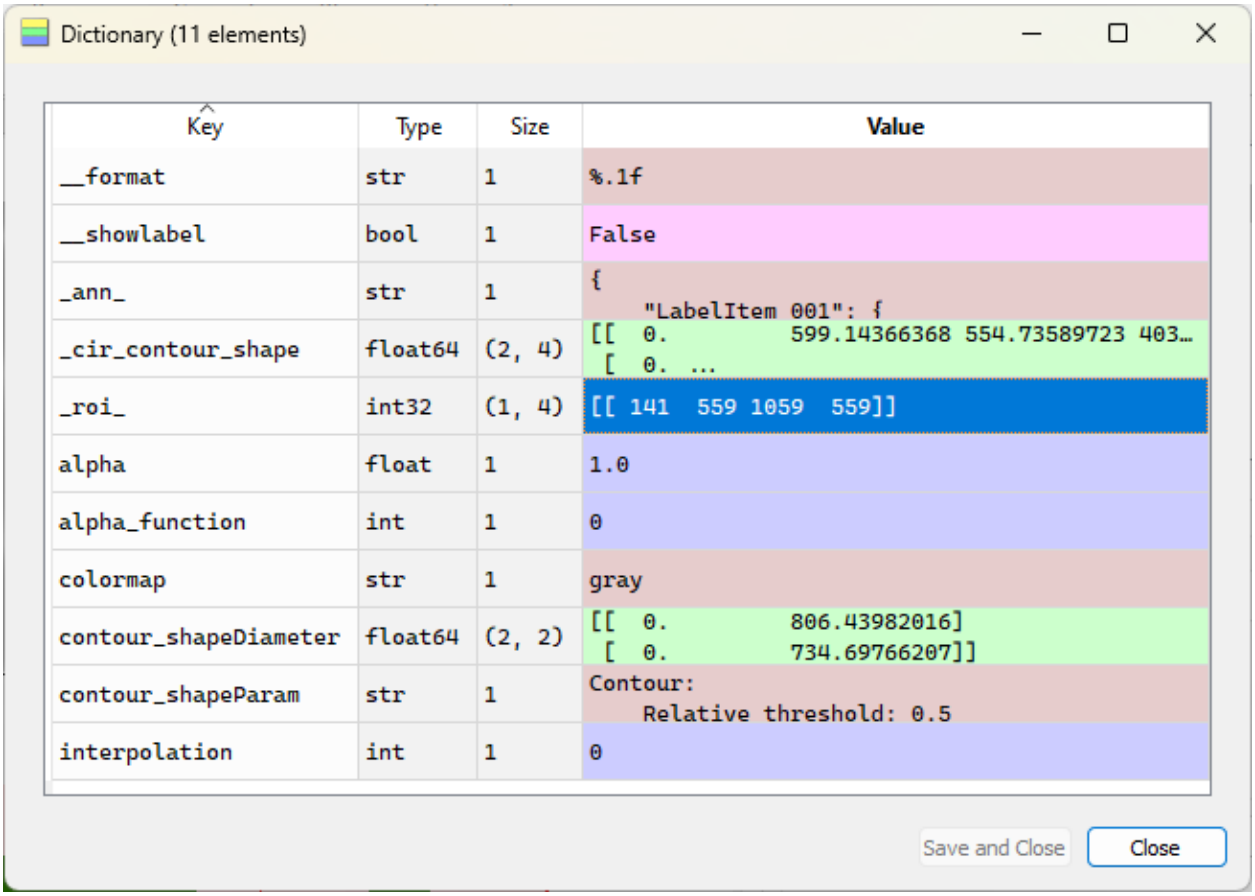


Fig. 62: The "Metadata" dialog opens. Among other information, it displays the annotations (in a JSON format), some style information (e.g. the colormap), and the ROI.

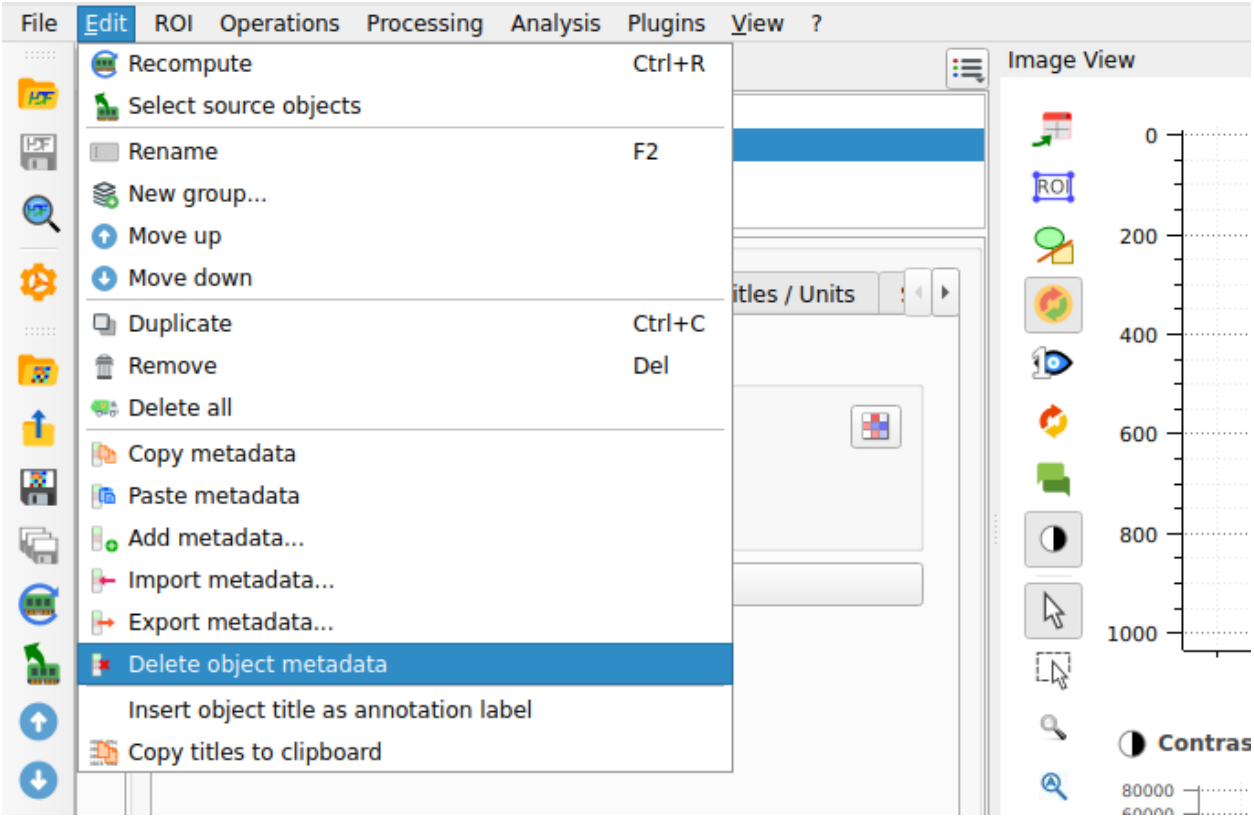


Fig. 63: Select the "Delete object metadata" entry in the "Edit" menu.

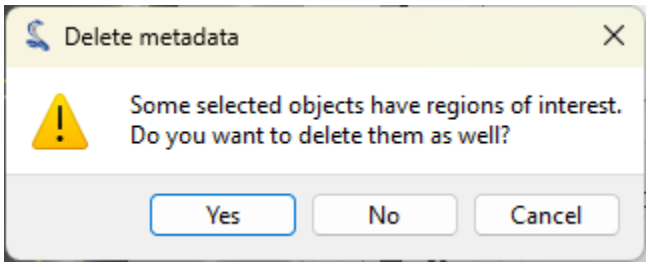


Fig. 64: The "Delete metadata" dialog. Click "No" to keep the ROI and delete the rest of the metadata.

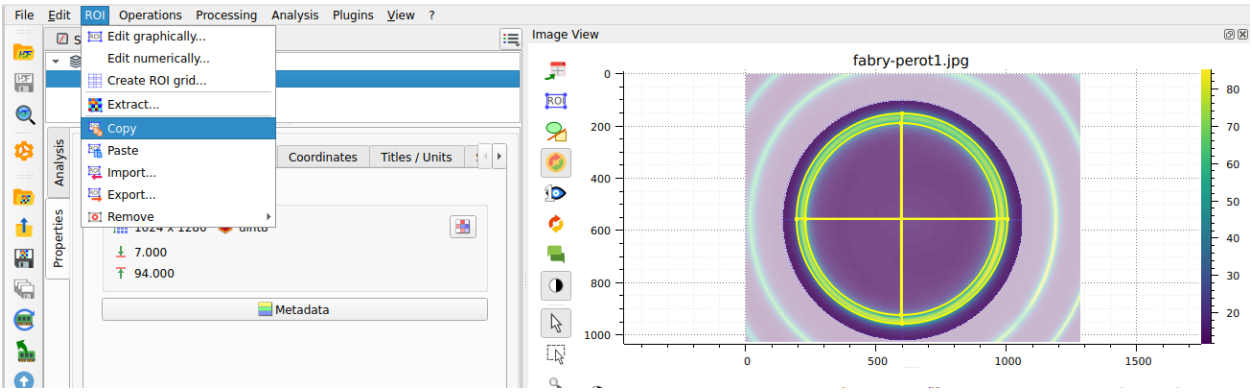


Fig. 65: The "Copy" entry in the "ROI" menu.

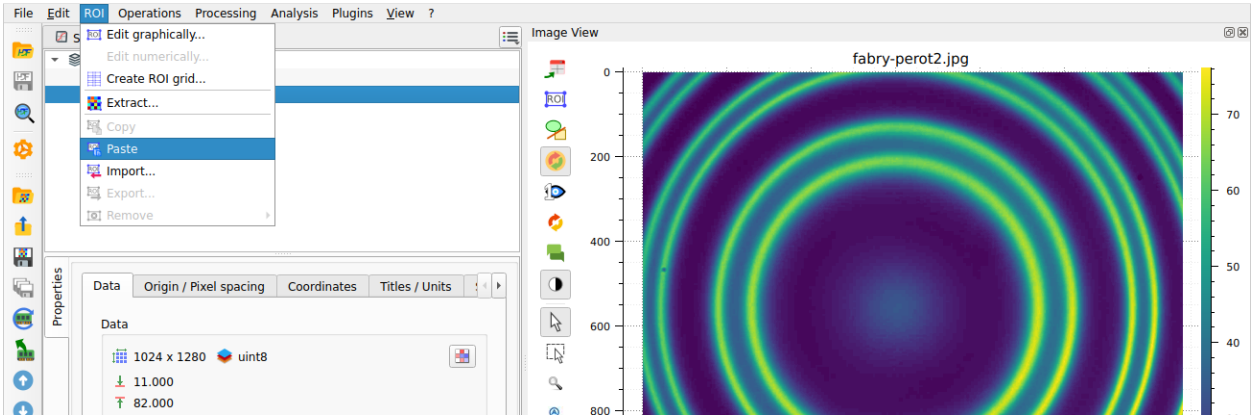


Fig. 66: The "Paste" entry in the "ROI" menu, or the button in the toolbar.

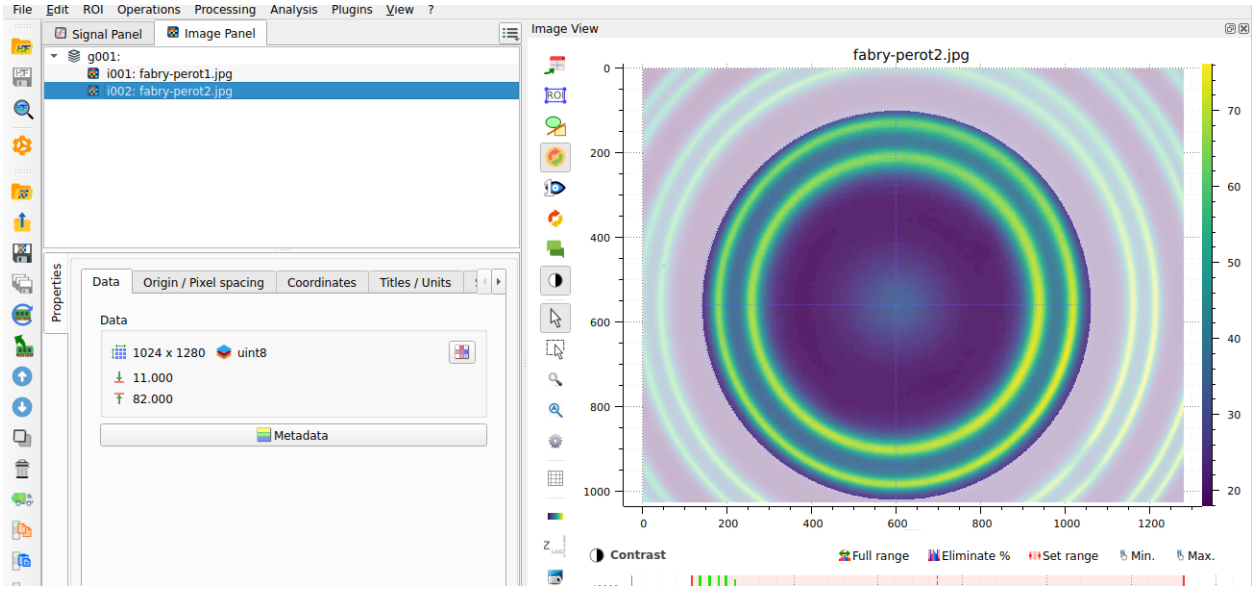


Fig. 67: The ROI is added to the second image.

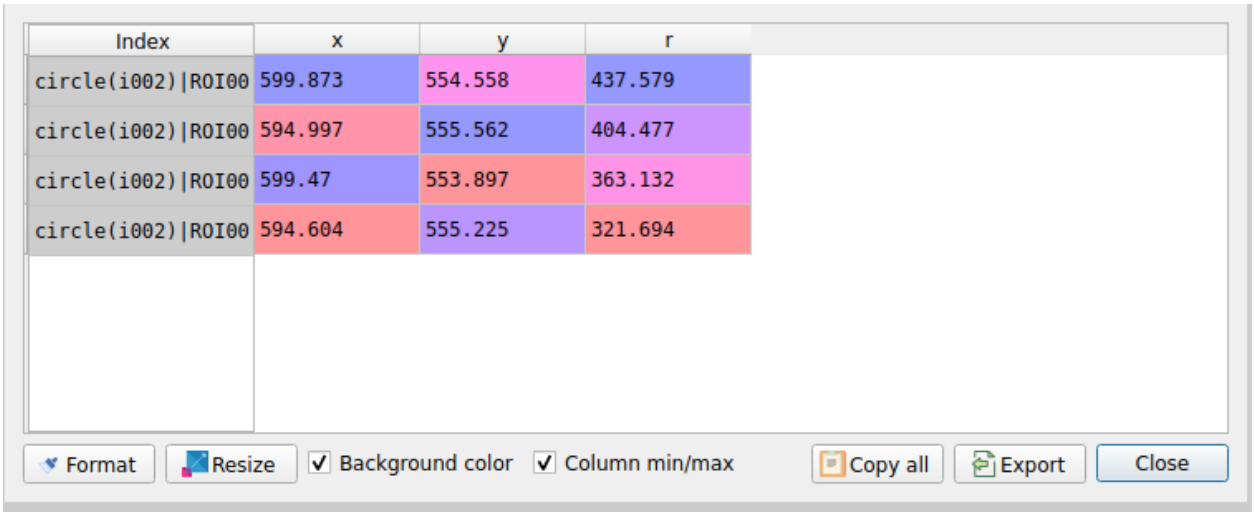


Fig. 68: The "Results" dialog for the second image.

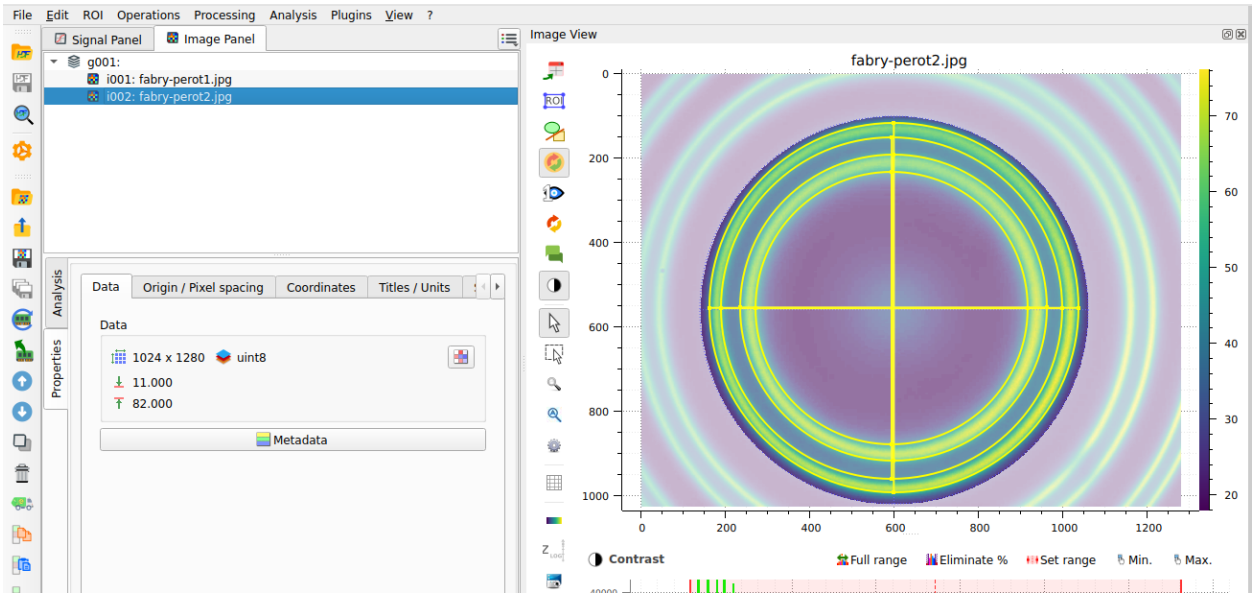


Fig. 69: The fitted circles are displayed on the image.

profile" dialog will open, letting you select the profile row (for a horizontal profile) or column (for a vertical profile) on the image. You can also enter the row/column number directly by clicking on "parameters…" in the "Line profile" dialog.

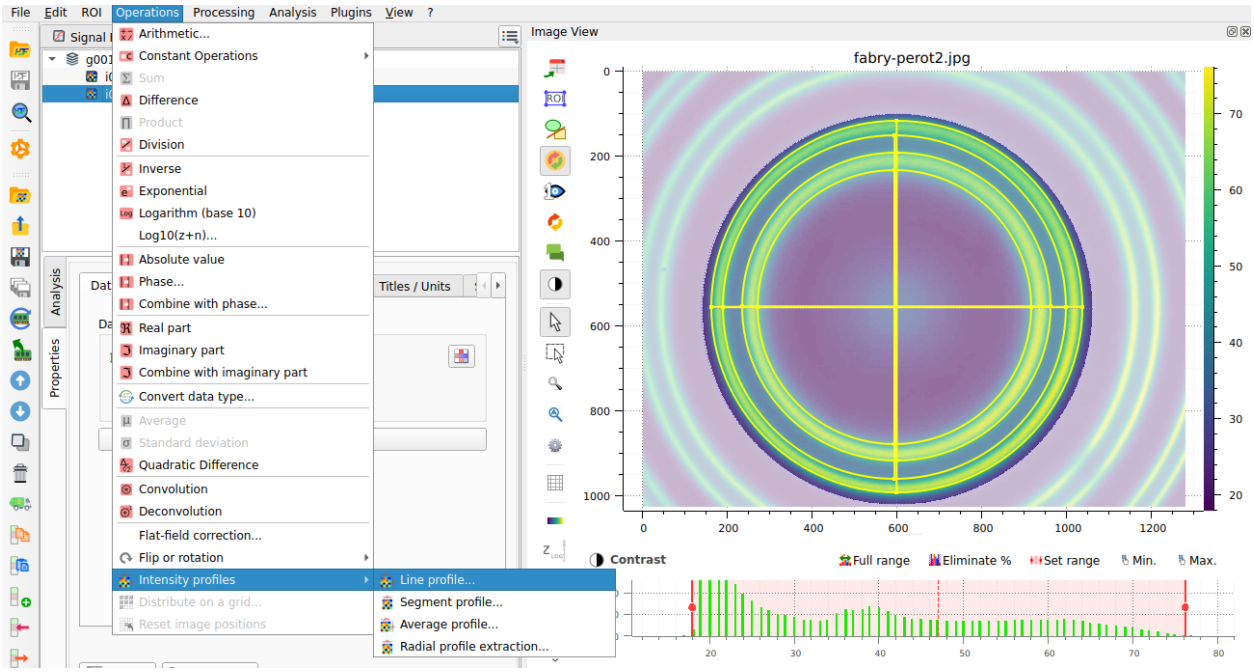


Fig. 70: Select the "Line profile…" entry in the "Operations" menu.

After clicking "OK", the intensity profile along the X axis is computed and displayed in the "Signals" panel. DataLab automatically switches to the "Signals" panel to display the profile.

If you want to do some measurements on the profile, or add annotations, you can open the signal in a separate window, by clicking on the "View in a new window" entry in the "View" menu (or the button in the toolbar).

Now, let's try the second option for extracting the intensity profile along the X axis, using the "Cross section" tool in the vertical toolbar on the left of the visualization panel (this tool is a PlotPy feature). Before being able to use it, we need to select the image again in the visualization panel (otherwise the tool is grayed out). Then, we can click on the image to display the intensity profiles along the X and Y axes. DataLab integrates a modified version of this tool which allows you to transfer the profile to the "Signals" panel for further processing. Select the "Cross section" tool in the vertical toolbar and click on the image to display the intensity profiles along the X and Y axes.

Click on the "Process signal" button in the toolbar near the profile to transfer the profile to the "Signals" panel. The intensity profile is added to the "Signals" panel, and DataLab switches to this panel to display the profile.

Save the workspace

Finally, we can save the workspace to a file. The workspace contains all the images and signals that were loaded or processed in DataLab. It also contains the analysis results, the visualization settings (colormaps, contrast, etc.), the metadata, and the annotations.

If you want to load the workspace again, you can use the "File > Open HDF5 file…" (or the button in the toolbar) to load the whole workspace, or the "File > Browse HDF5 file…" (or the button in the toolbar) to load only a selection of data sets from the workspace.

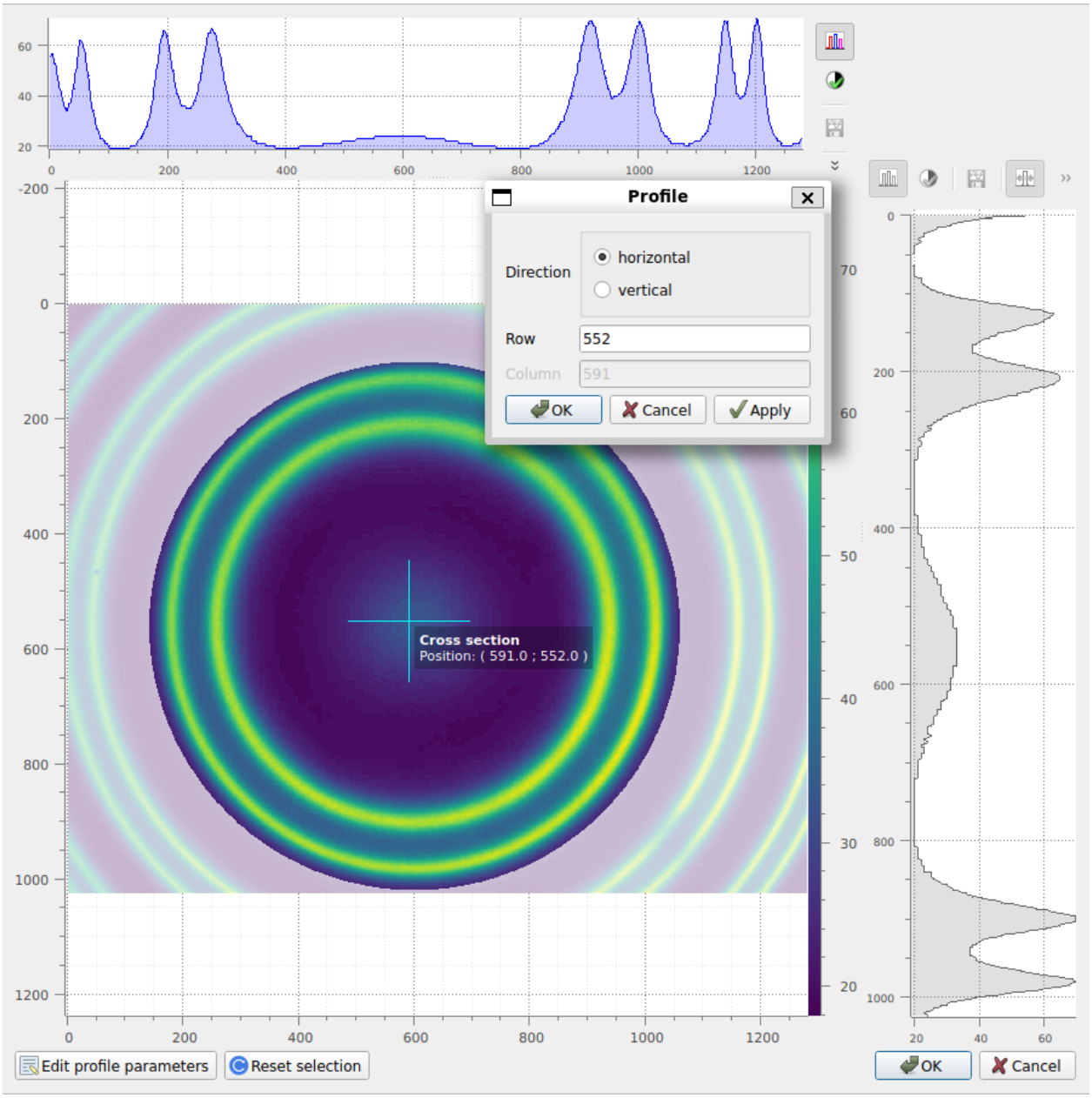


Fig. 71: The "Profile" dialog opens. Enter the row of the horizontal profile (or the column of the vertical profile) in the dialog box that opens. Click "OK".

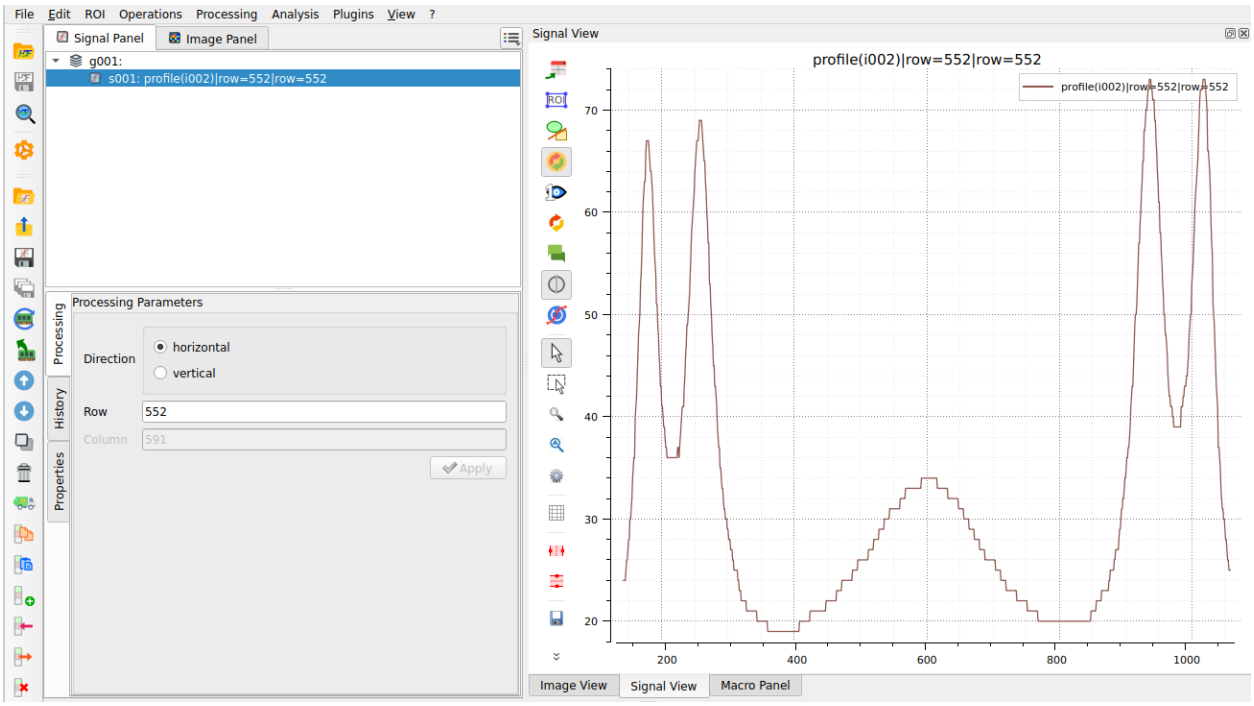


Fig. 72: The intensity profile is added to the "Signals" panel, and DataLab switches to this panel to display the profile.

Measuring Laser Beam Size

This example shows how to measure the size of a laser beam along the propagation axis using DataLab:

- Load all the images in a folder
- Apply a threshold to the images
- Extract the intensity profile along an horizontal line
- Fit the intensity profile to a Gaussian function
- Compute the full width at half maximum (FWHM) of intensity profile
- Try another method: extract the radial intensity profile
- Compute the FWHM of the radial intensity profile
- Perform the same analysis on a stack of images and on the resulting profiles
- Plot the beam size as a function of the position along the propagation axis

Load the images

Note: The images used in this tutorial "TEM00_z_*.jpg" are available in the tutorial data folder of DataLab's installation directory (`<DataLab installation directory>/data/tutorials/`).

Alternatively, they can be downloaded from the online documentation at https://datalab-platform.com.

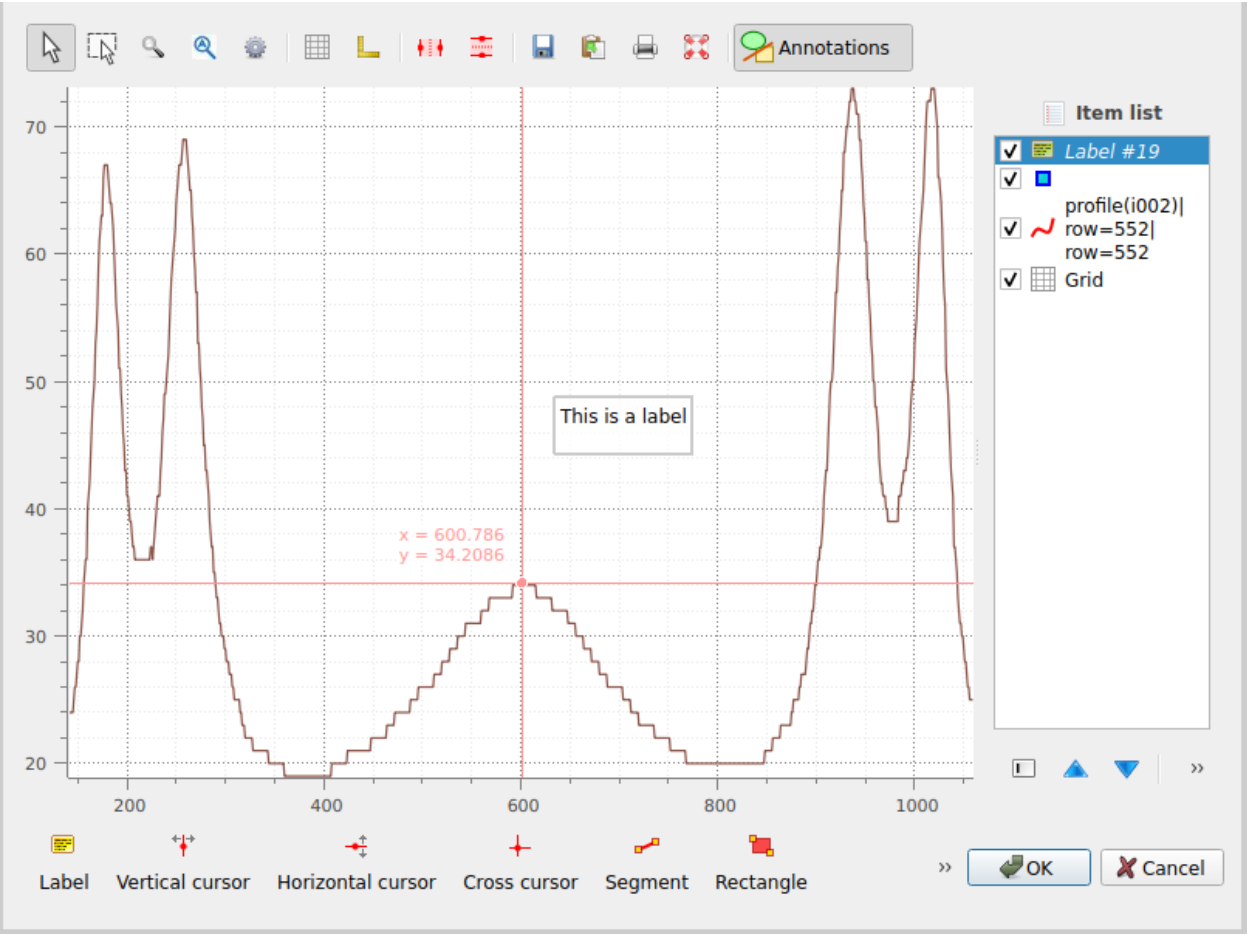


Fig. 73: The signal is displayed in a separate window. Here, we added vertical cursors and a text label. As for the images, the annotations are stored in the metadata of the signal and saved together with the signal data when the workspace is saved. Click on "OK" to close the window.

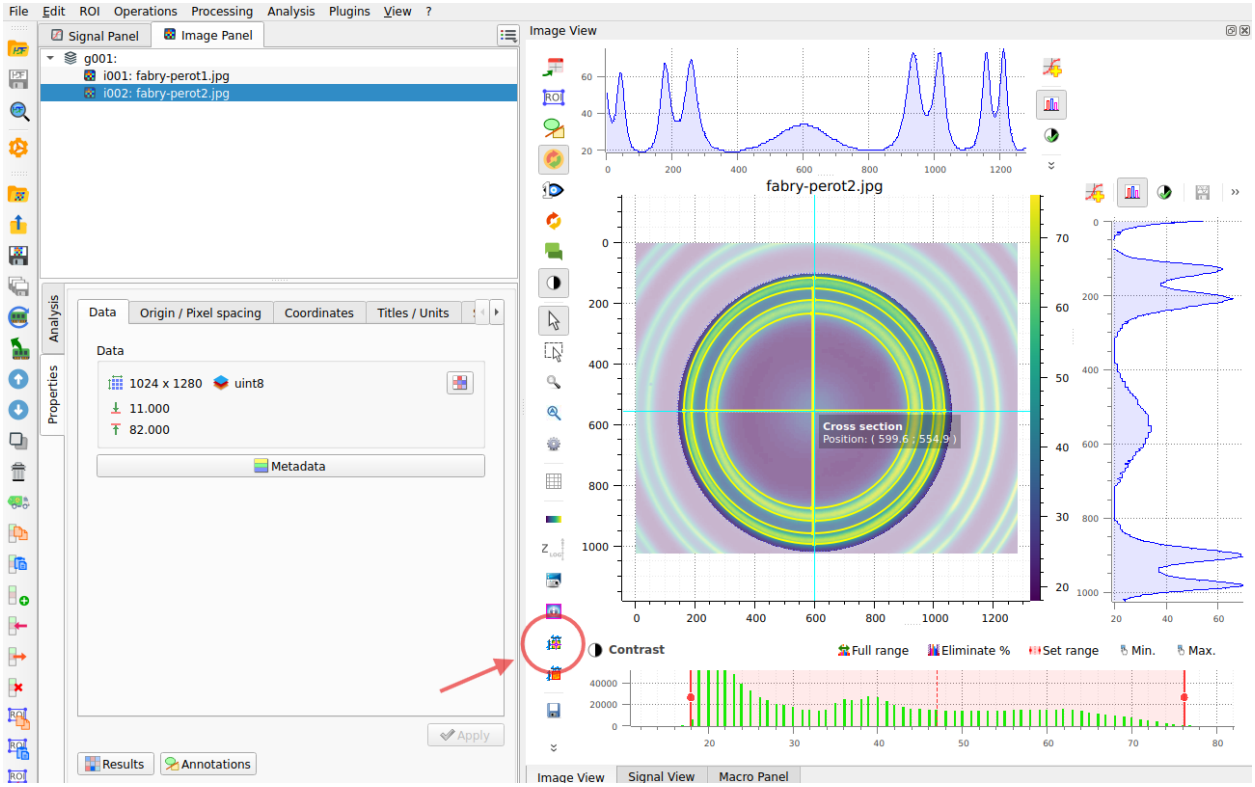


Fig. 74: The cross section tool in the view panel. In the red circle, the button to activate it. In the blue circle, the button to transfer the profile to the "Signals" panel.

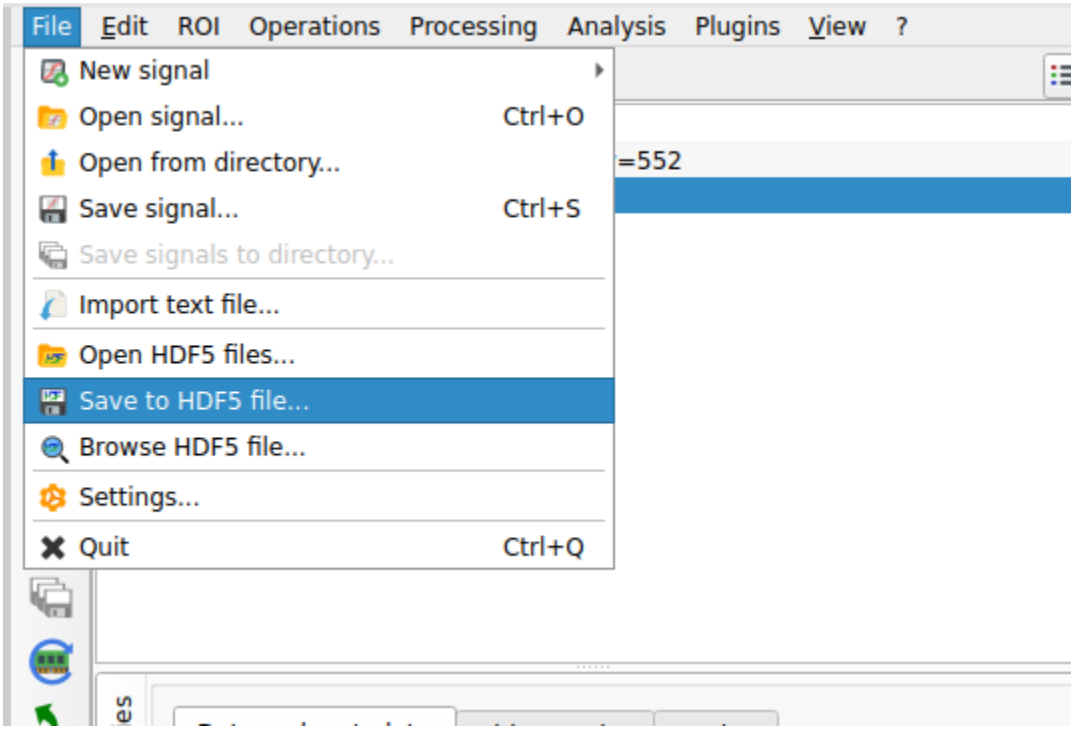


Fig. 75: Save the workspace to a file with "File > Save to HDF5 file…", or the button in the toolbar.

First, we open DataLab and load the images. When working with multiple images, the most efficient approach is to use the "Open from directory…" option from the "File" menu (or click the button in the toolbar). This feature loads all images from a selected directory at once.

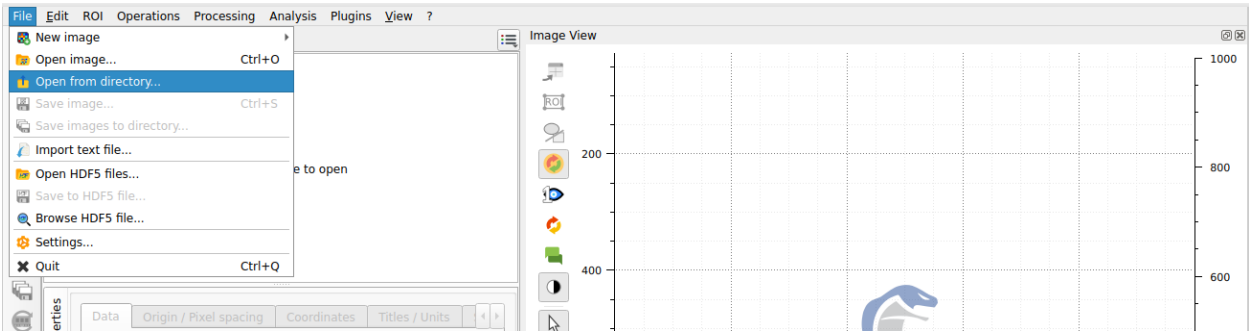


Fig. 76: The "File > Open Image…" menu

The images are now loaded in the "Image panel". You can zoom in and out by right-clicking and dragging the mouse vertically. To pan the image, use the middle mouse button while dragging.

Note: To view multiple images simultaneously, select the image group and choose "View images side-by-side" from the "View" menu.

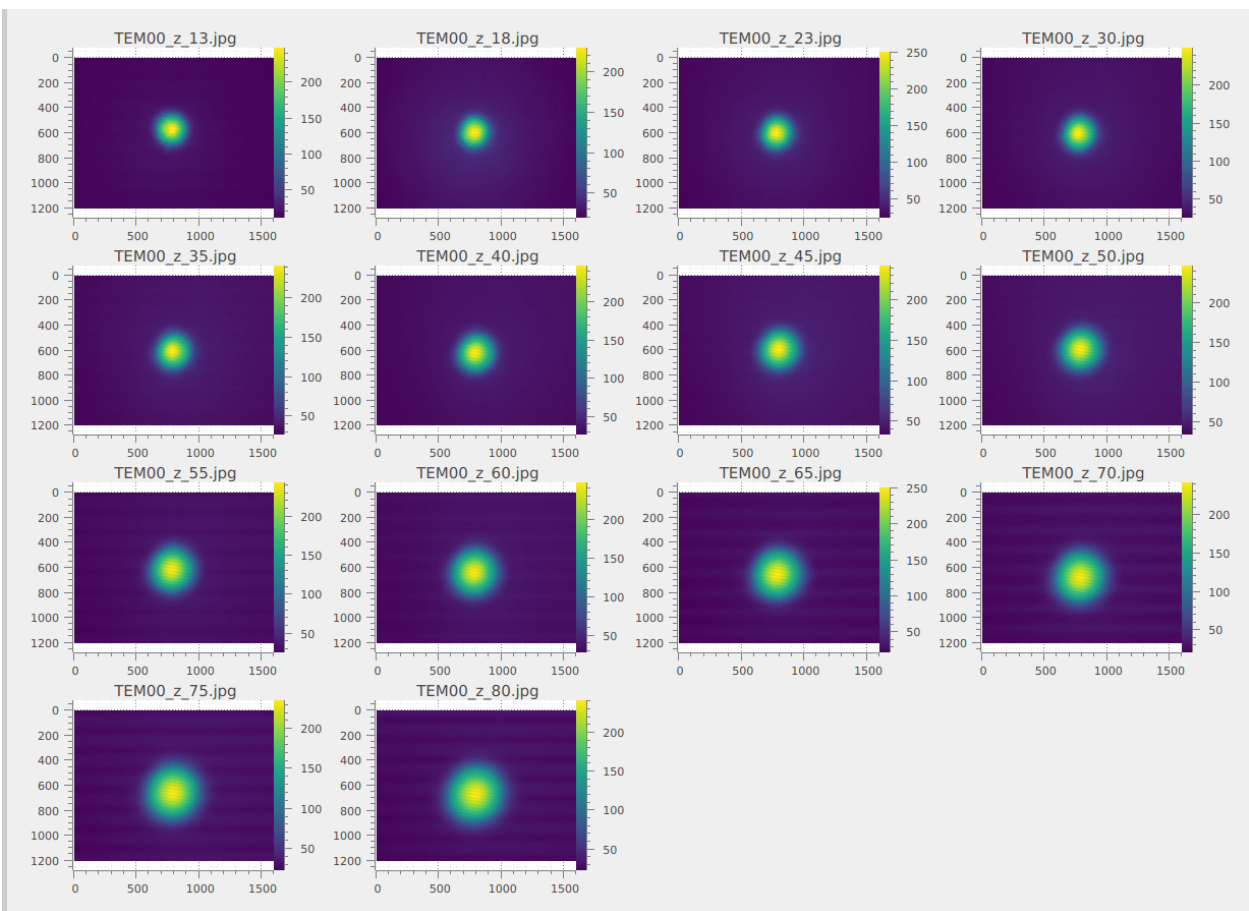


Fig. 79: Viewing images side by side.

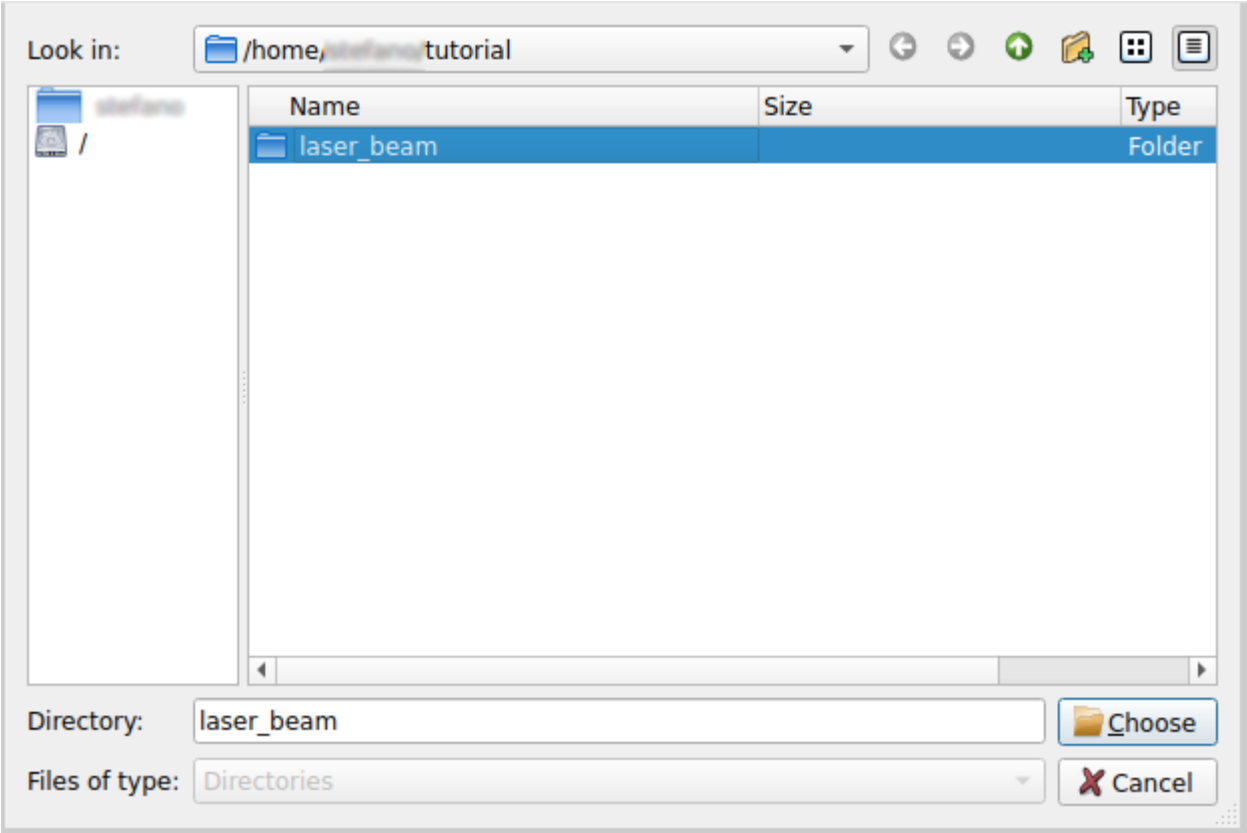


Fig. 77: Select the folder containing the images and click "Choose".

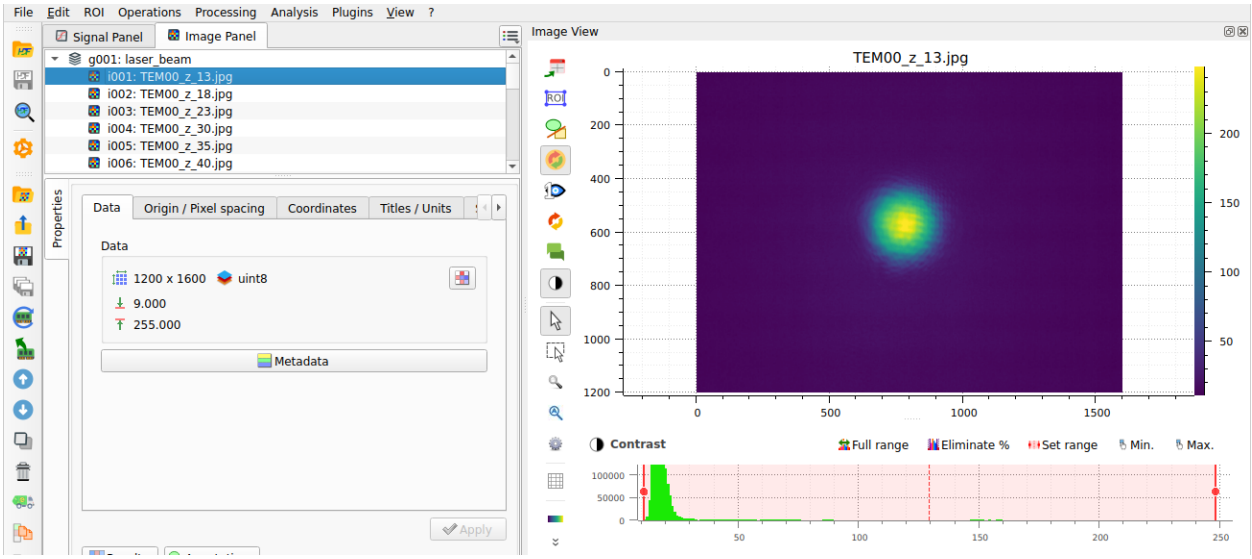
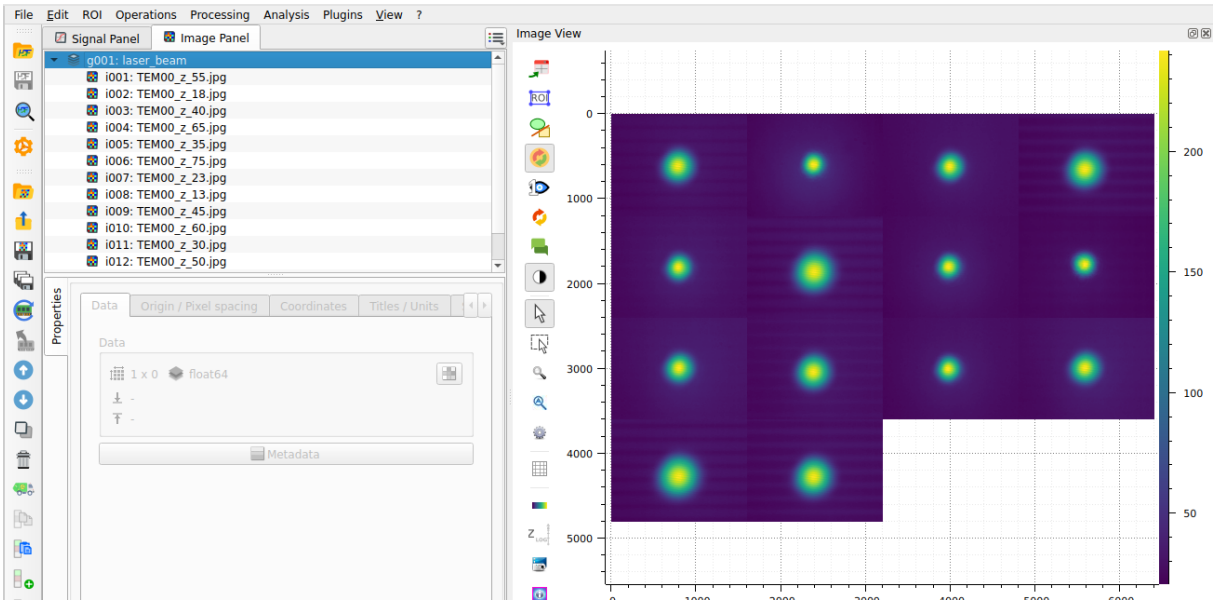


Fig. 78: Zoom in and out with the right mouse button. Pan the image with the middle mouse button.

Warning: The "Processing > Geometry" menu includes a "Distribute on a grid" option . This feature repositions the images by applying offsets to their X and Y coordinates, arranging them in a grid layout for side-by-side viewing. Note that this operation modifies the image coordinates.



Images distributed on 4 columns grid.

To restore the original image positions, use the "Reset image positions" option from the "Processing > Geometry" menu. Note that this operation sets all image origins to match the first image's origin, which means any initial differences in image origins will be lost.

Remove background noise

When we select one of the images, we notice the presence of background noise, making it beneficial to apply a threshold to the images. Several methods are available to estimate the background noise level.

One approach utilizes the "Cross section" tool, which is provided by the *PlotPy <https://github.com/PlotPyStack/plotpy>* library that DataLab uses for signal and image visualization. Select an image from the "Image panel", choose the corresponding image in the visualization panel, and activate the "Cross section" tool from the vertical toolbar on the left side of the visualization panel. This reveals that the background noise level is approximately 30 lsb.

Another method for measuring background noise, also provided by *PlotPy <https://github.com/PlotPyStack/plotpy>*, involves using the "Image statistics" tool from the vertical toolbar on the left side of the visualization panel. This tool displays statistical information for a rectangular region that you define by dragging the mouse across the image. This analysis confirms that the background noise level is approximately 30 lsb.

Note that these tools are not persistent: the analysis results disappear when you select another image, they are intended to provide a fast insight on the image data.

Now we can clip the image at 35 lsb to remove the background noise using the "Processing > Level Adjustment > Clipping…" menu.

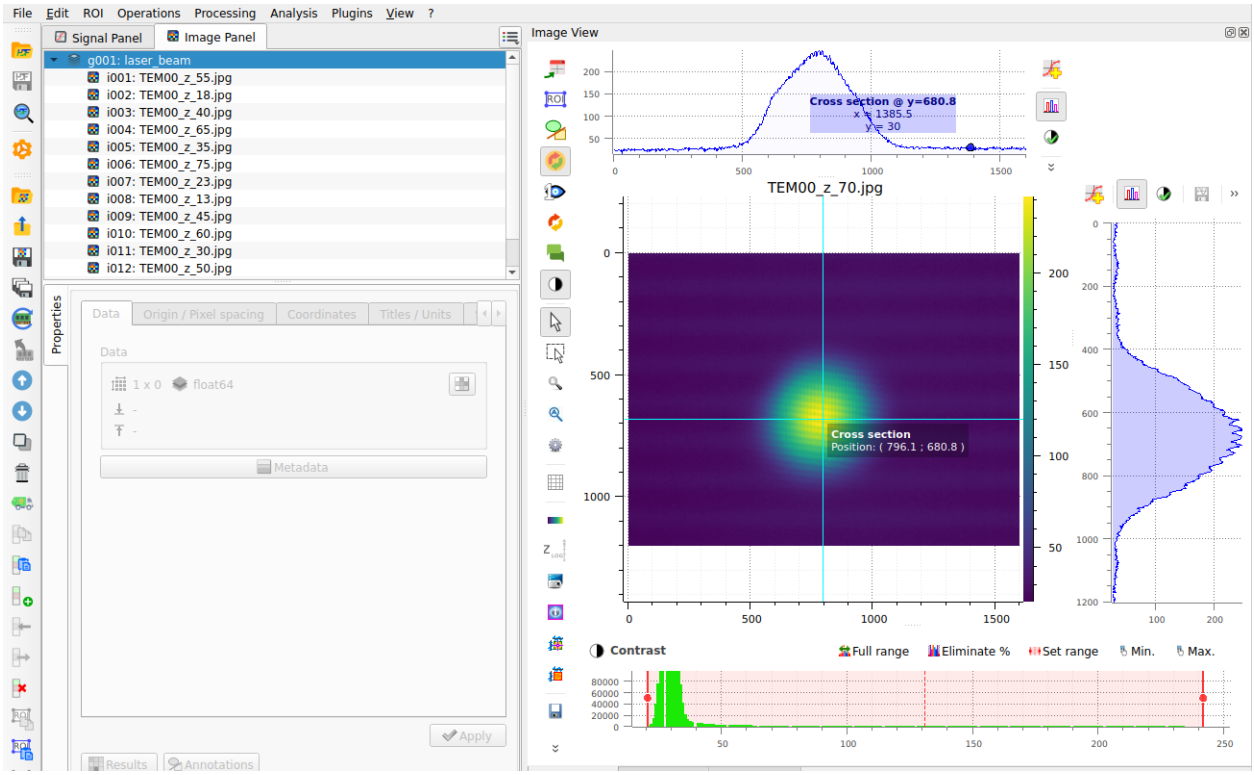


Fig. 80: An image from the "Image panel". To display the curve marker, select the profile curve and right-click to open the context menu, then choose "Markers > Bound to active item".

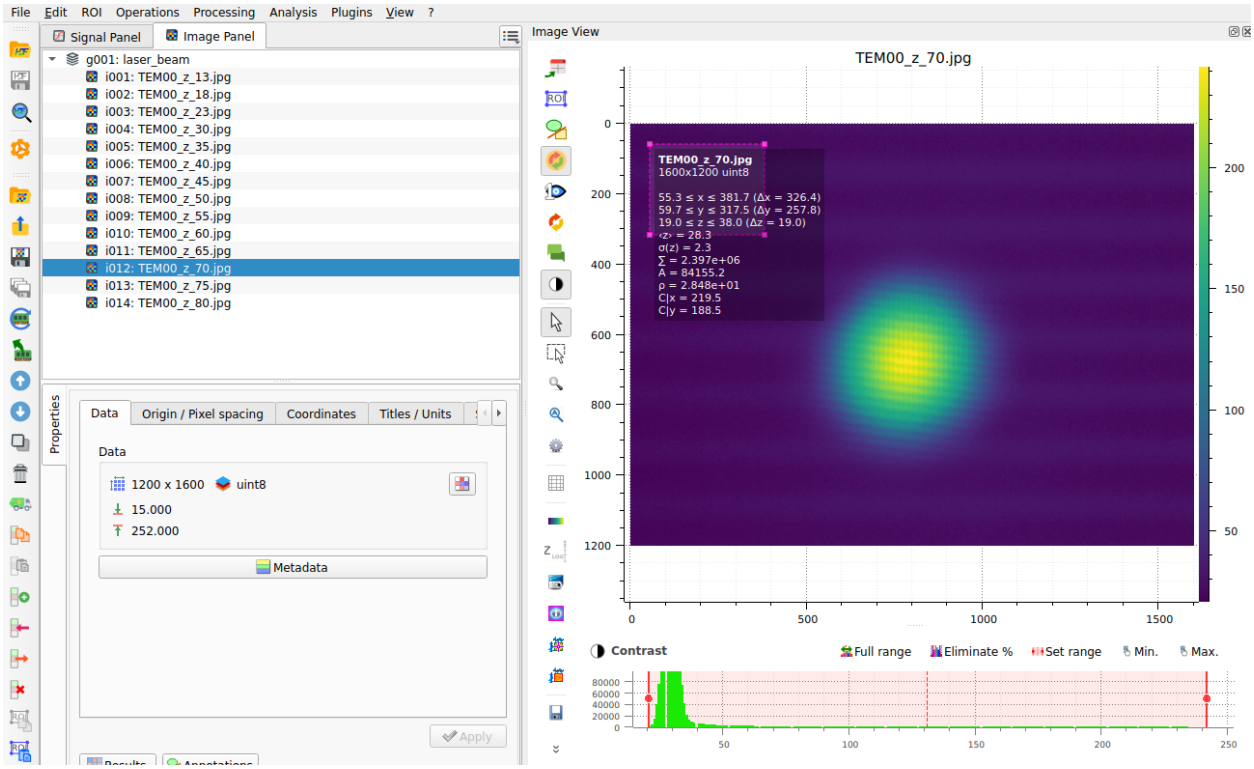


Fig. 81: The "Image statistics" tool in the vertical toolbar.

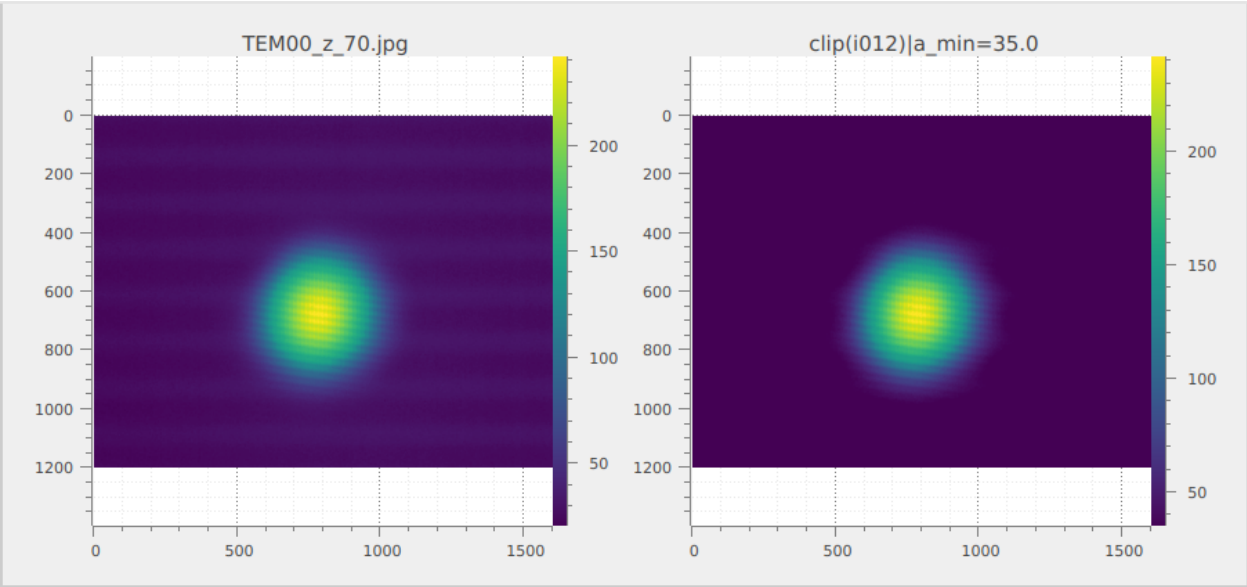


Fig. 82: The two original and the clipped images are displayed side by side in the "Image view" (using the "Distribute on a grid" feature seen previously).

Beam size measurement

We can now compute the centroid of the beam—that is, the position of its center of mass. To do this, select "Analysis > Centroid" from the menu.

Next, we can extract a line profile along the horizontal axis using "Analysis > Intensity profiles > Line profile". Set the row position to the previously computed centroid position (i.e., 668) using the "Set Parameters" button. See *Measuring Fabry-Perot fringes* for more details on intensity profile extraction.

The intensity profile will be displayed in the "Signal panel". We can then fit the profile to a Gaussian function using "Processing > Fitting > Gaussian fit". Here we have selected both signals for comparison.

Now let's explore another method to compute the FWHM. Returning to the intensity profile signal, we can directly compute the FWHM using "Analysis > Full width at half maximum". You can choose the estimation method based on your curve characteristics and optionally specify the interval to consider for this computation. Once complete, the results window will display the FWHM value, which is stored in the metadata and shown on the curve.

Let's also try another method to measure the beam size by returning to the image.

From the "Image panel", we can extract the radial intensity profile using "Analysis > Intensity profiles > Radial profile". The radial intensity profile can be computed around the centroid position, the center of the image, or a user-defined position, depending on your data. For our data, which appears to have radial symmetry with a center not necessarily identical to the image center, the best option is the centroid position.

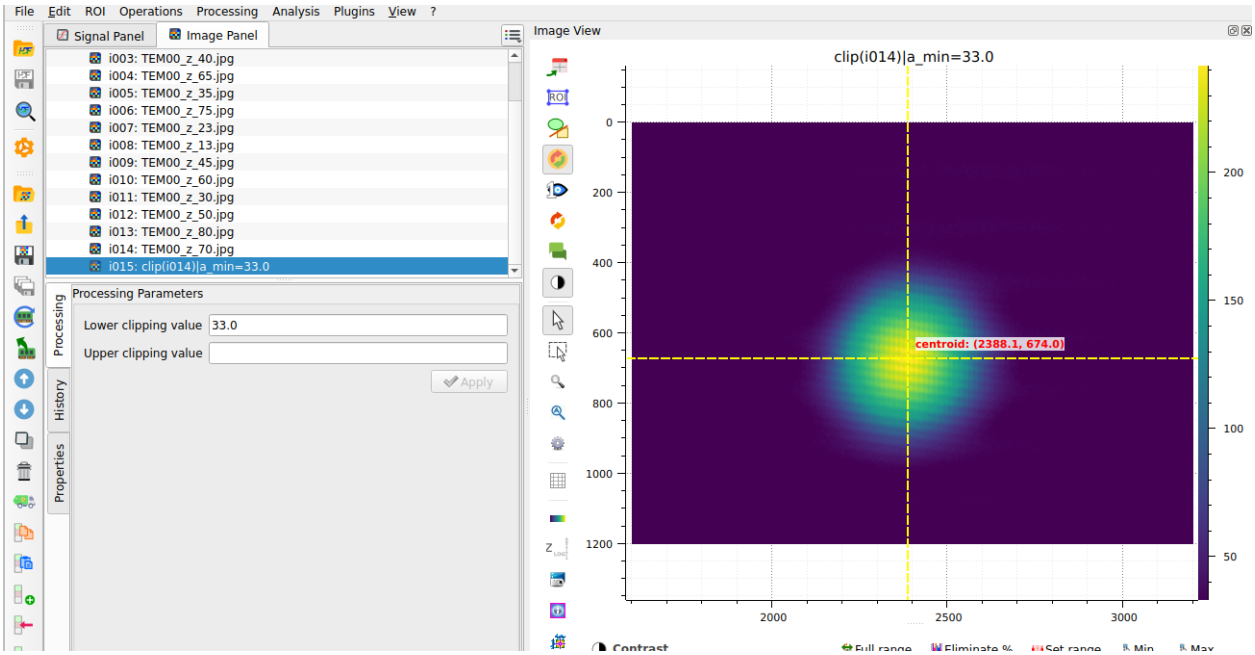


Fig. 83: The centroid position is displayed on the image.

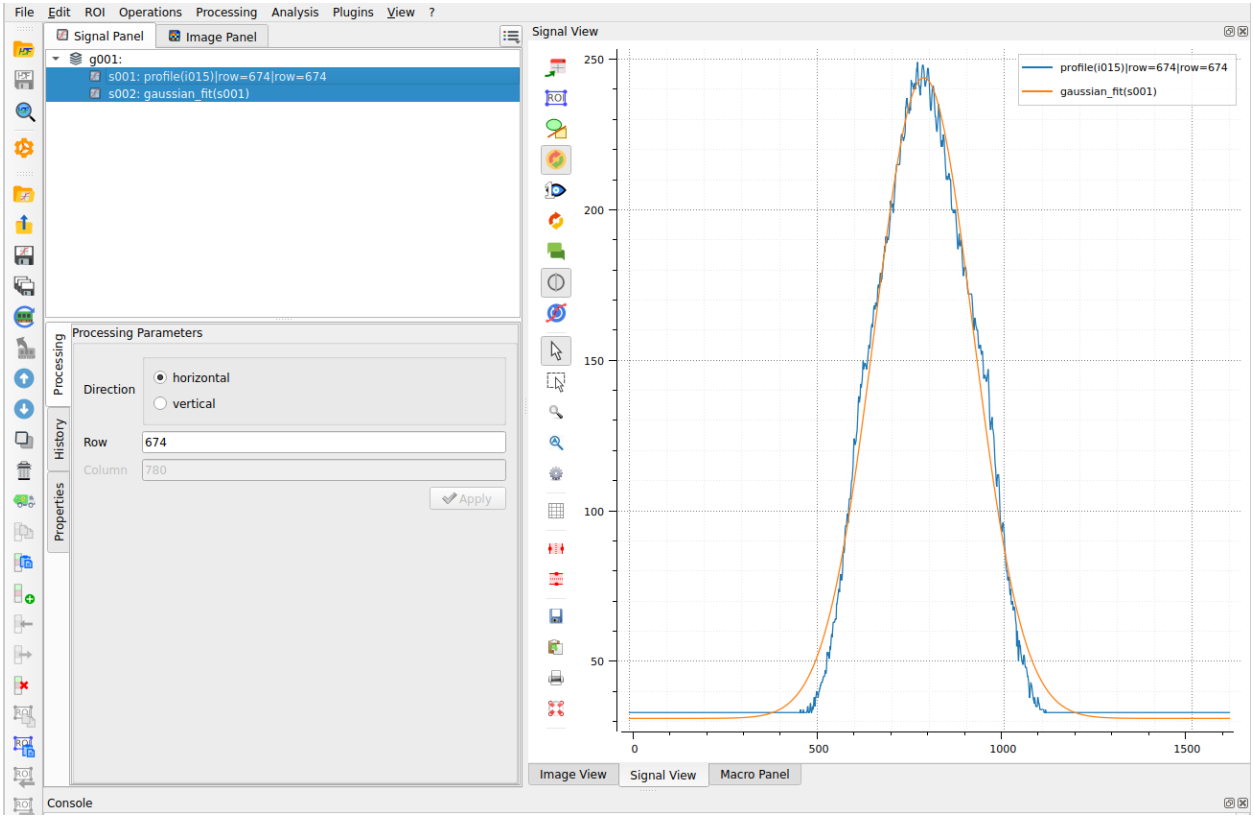


Fig. 84: The intensity profile fitted to a Gaussian function. Here both signals are selected.

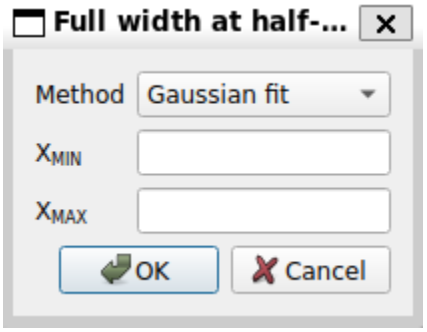


Fig. 85: The popup that allows to choose the method to estimate the FWHM.

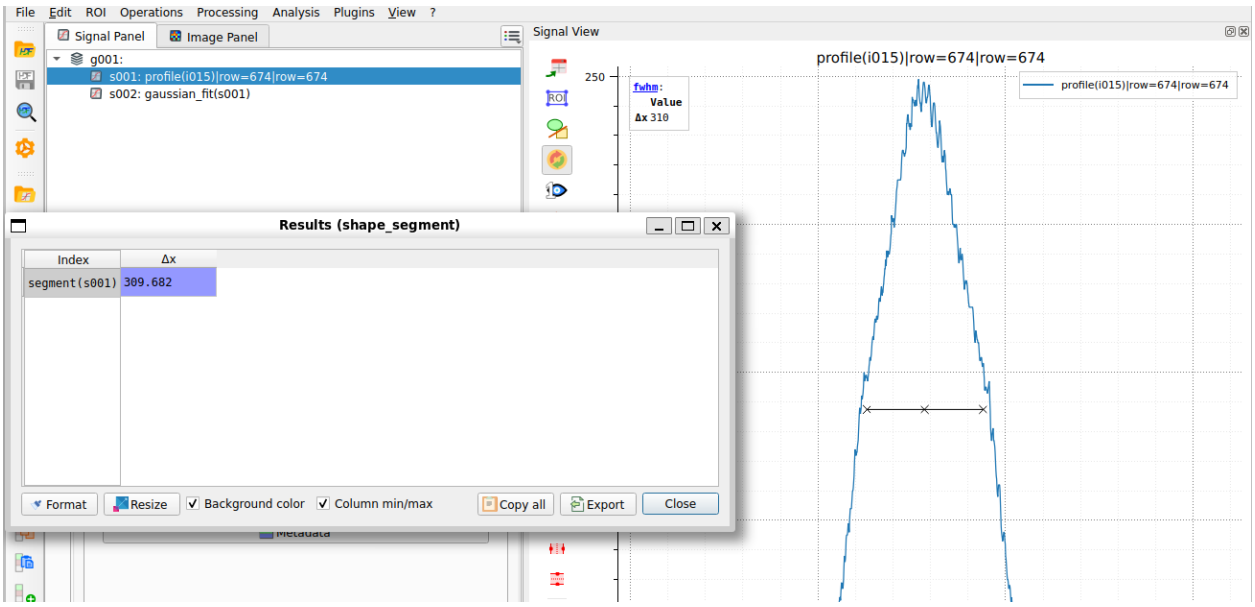


Fig. 86: The FWHM result shown in the popup and over the curve.

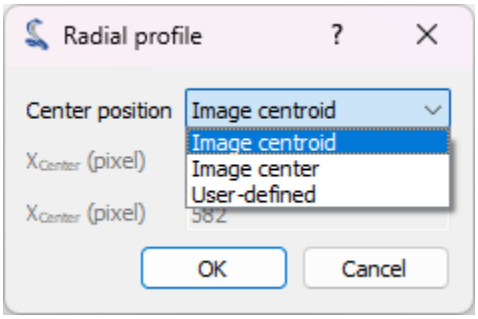


Fig. 87: The options available for the Radial Profile computation.

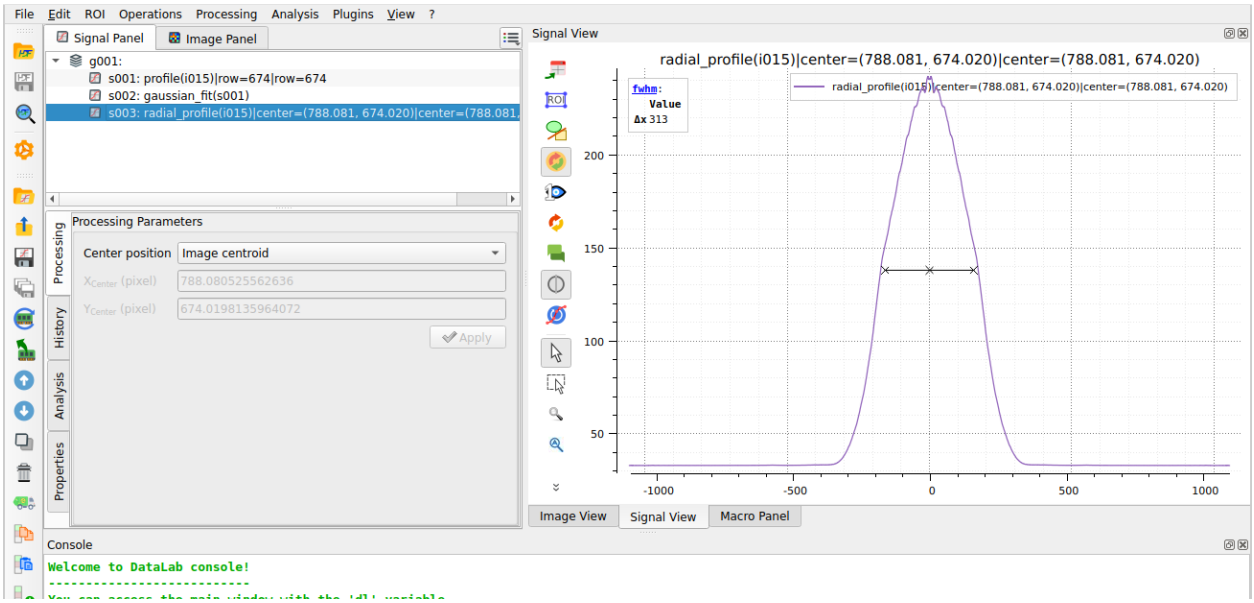


Fig. 88: The radial intensity profile displayed in the "Signal panel". It is smoother than the line profile, because it is computed from a larger number of pixels, thus averaging the noise.

Apply these operations to all the images

All the operations and computations performed on a single image can be applied to all images in the "Image panel".

To begin, clean the "Signal panel" using "Edit > Delete all" or the button in the toolbar. Also remove intermediate results from the "Image panel" by selecting the images created during prototyping and deleting them individually using "Edit > Remove" or the button.

Next, select all images in the "Image panel" (individually or by selecting the entire group "g001").

Apply the clipping operation to all images, then extract the radial intensity profiles for all images (after selecting the entire group "g002"—it should be automatically selected if you had "g001" selected before applying the threshold).

We can now compute the FWHM for all radial intensity profiles. The "Results" dialog will display the FWHM values for all profiles.

Note: To display the analysis results again, select "Show results" from the "Analysis" menu, or click the "Show results" button below the image list:

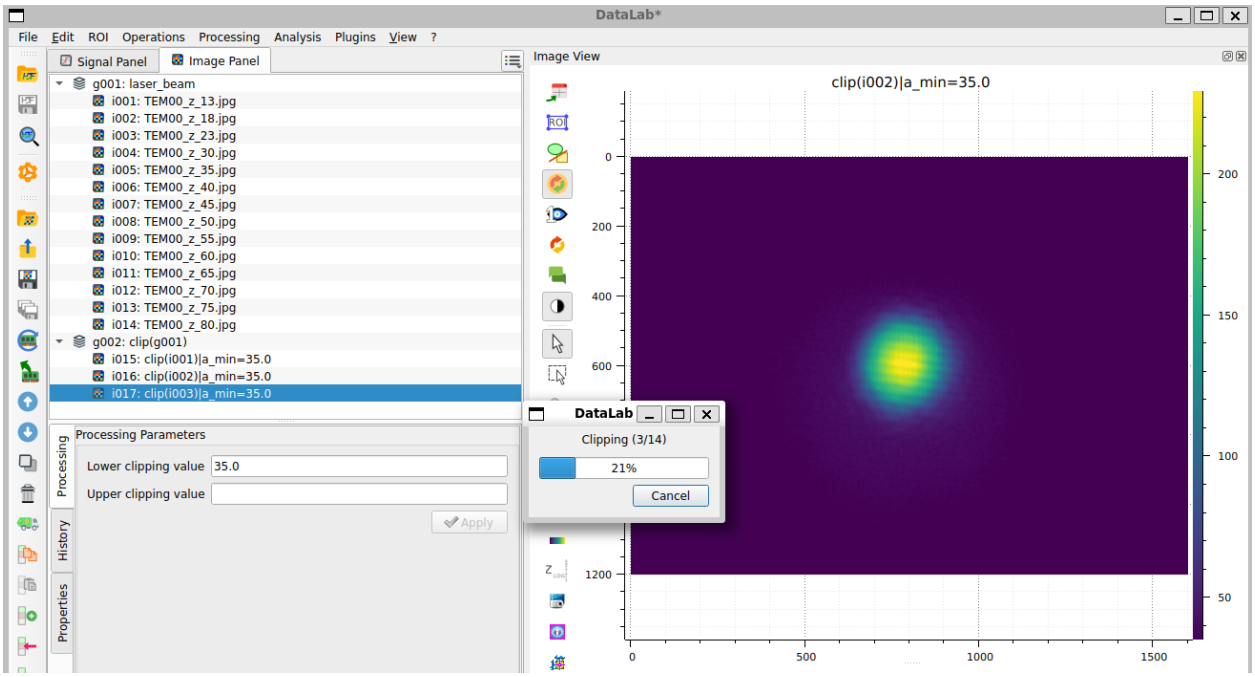


Fig. 89: The clipping applies to all images of the group.

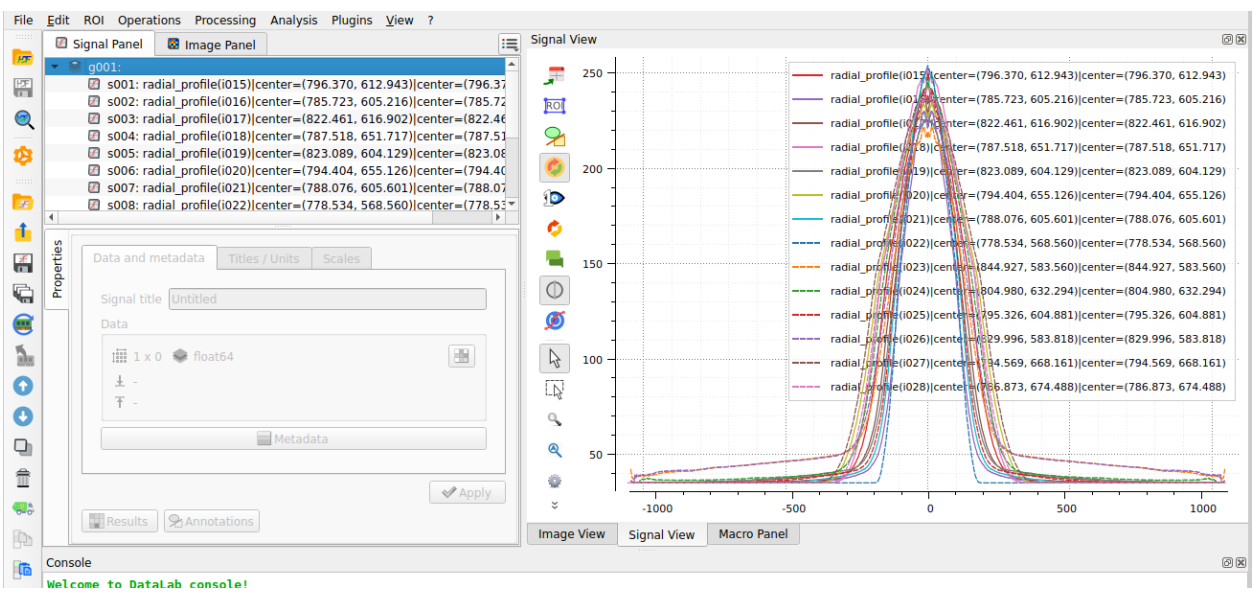
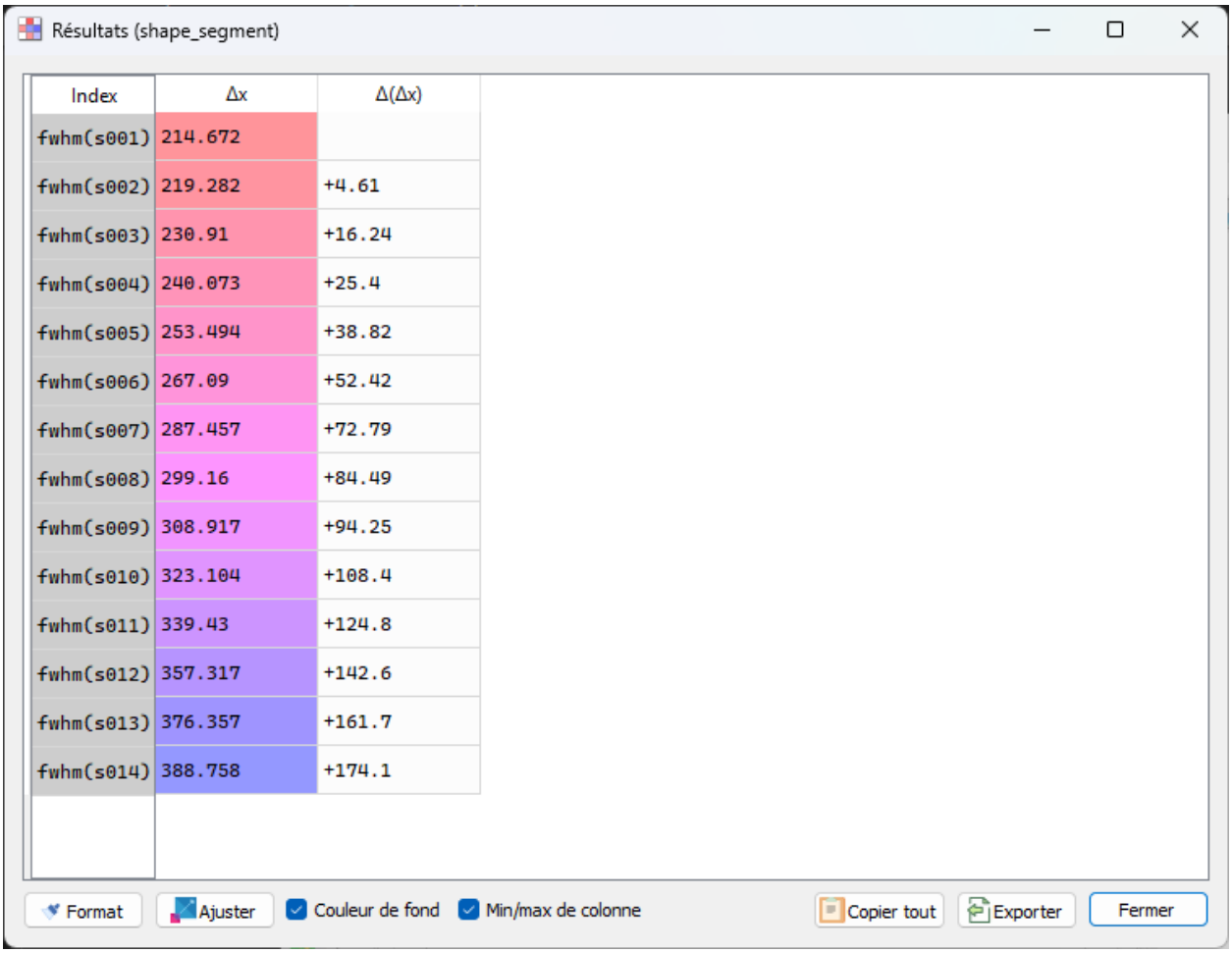


Fig. 90: The “Signal panel” now contains all the radial intensity profiles.



Index	Δx	Δ(Δx)
fwhm(s001)	214.672	
fwhm(s002)	219.282	+4.61
fwhm(s003)	230.91	+16.24
fwhm(s004)	240.073	+25.4
fwhm(s005)	253.494	+38.82
fwhm(s006)	267.09	+52.42
fwhm(s007)	287.457	+72.79
fwhm(s008)	299.16	+84.49
fwhm(s009)	308.917	+94.25
fwhm(s010)	323.104	+108.4
fwhm(s011)	339.43	+124.8
fwhm(s012)	357.317	+142.6
fwhm(s013)	376.357	+161.7
fwhm(s014)	388.758	+174.1

Fig. 91: The “Results” dialog displays the FWHM values for all the profiles.

Finally, we can plot the beam size as a function of position along the propagation axis using the "Plot results" feature from the "Analysis" menu. This feature allows you to plot result datasets by selecting the x and y axes from the available result columns. Here, we will plot the FWHM values (L) as a function of the image index (*Indices*).

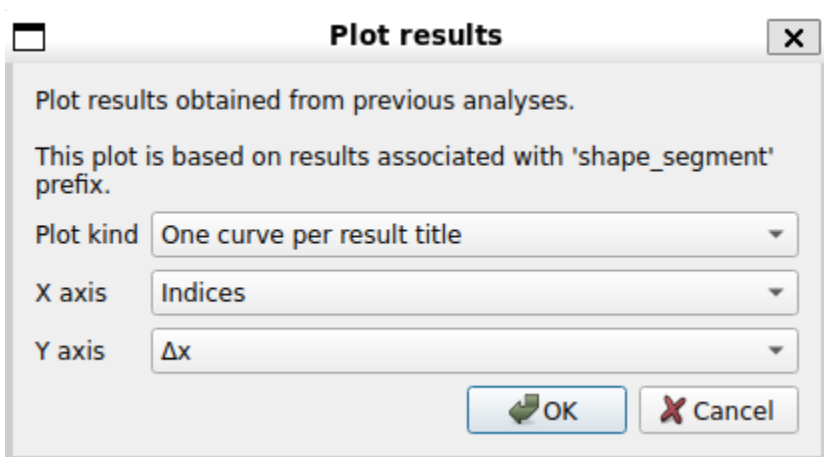


Fig. 92: The "Plot results" feature in the "Analysis" menu.

We can also calibrate the X and Y axes using "Processing > Linear calibration". Here we set the X axis to the position in mm (entering the title and unit in the "Properties" group box) using the formula: $X[\text{mm}] = 0.5 \cdot i + 10$ where i is the image index.

Finally, we can save the workspace to a file . The workspace contains all the images and signals that were loaded or processed in DataLab. It also contains the analysis results, the visualization settings (colormaps, contrast, etc.), the metadata, and the annotations.

If you want to load the workspace again, you can use the "File > Open HDF5 file…" (or the button in the toolbar) to load the whole workspace, or the "File > Browse HDF5 file…" (or the button in the toolbar) to load only a selection of data sets from the workspace.

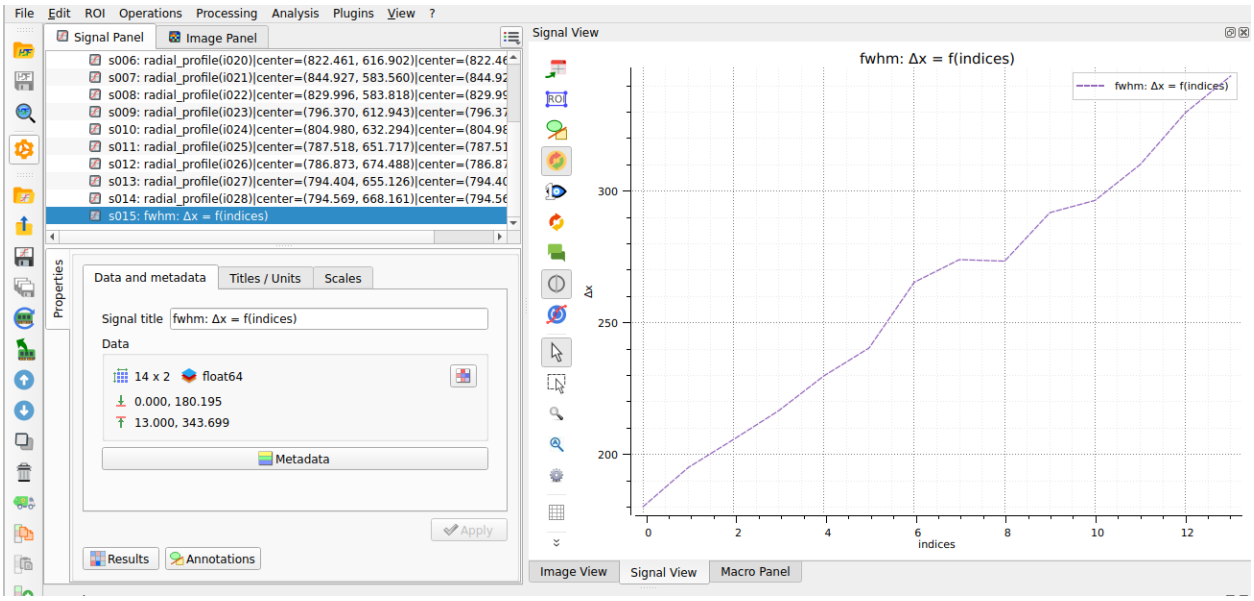


Fig. 93: The beam size as a function of position along the propagation axis (the position is in arbitrary units—the image index).

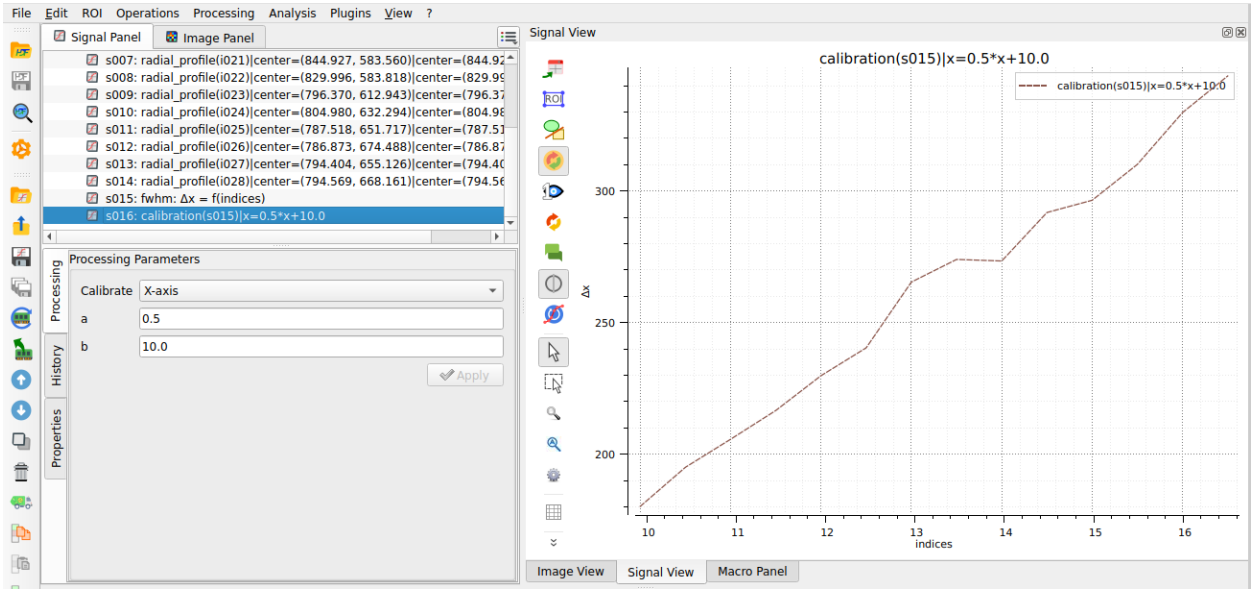


Fig. 94: The calibrated beam size as a function of the position along the propagation axis (the position is now in mm).

Prototyping a custom processing pipeline

For your specific application, it is possible that you will need a function that is not available in DataLab by default. In this case, you can define your own function and integrate it in DataLab to conveniently analyze your data. DataLab has been designed to be flexible and extensible, and the goal of this tutorial is to show you how to do that.

This example shows how to prototype a custom image processing pipeline using DataLab:

- Define a custom processing function
- Create a macro-command to apply the function to an image
- Use the same code from an external IDE (e.g. Spyder) or a Jupyter notebook
- Create a plugin to integrate the function in the DataLab GUI

Define a custom processing function

To illustrate the extensibility of DataLab, we will use a simple image processing function that is not available in the standard DataLab distribution, and that represents a typical use case for prototyping a custom processing pipeline.

The function that we will work on is a denoising filter that combines the ideas of averaging and edge detection. This filter will average the pixel values in the neighborhood, but with a twist: it will give less weight to pixels that are significantly different from the central pixel, assuming they might be part of an edge or noise. Once you have understood how to implement custom processing in DataLab, you will define your own functions adapted to your specific needs.

Here is the code of the `weighted_average_denoise` function:

```
def weighted_average_denoise(data: np.ndarray) -> np.ndarray:
    """Apply a custom denoising filter to an image.

    This filter averages the pixels in a 5x5 neighborhood, but gives less weight
    to pixels that significantly differ from the central pixel.
    """

    def filter_func(values: np.ndarray) -> float:
        """Filter function"""
        central_pixel = values[len(values) // 2]
        differences = np.abs(values - central_pixel)
        weights = np.exp(-differences / np.mean(differences))
        return np.average(values, weights=weights)

    return spi.generic_filter(data, filter_func, size=5)
```

Setup the test environment

To test our processing function, we will use an image generated from a DataLab plugin example (*plugins/examples/datalab_example_imageproc.py*). Before starting, make sure that the plugin is installed in DataLab (see the first steps of the tutorial *Detecting blobs on an image* to understand how to do it).

We then reorganize the DataLab window layout to have a comfortable environment for developing and testing our function: we reorganize the window layout of DataLab to have the "Image Panel" on the left and the "Macro Panel" on the right. To do so, you can simply drag and drop the panels to the desired position. If the "Macro Panel" is not visible, you can enable it from the "View > Panels > Macro Panel".

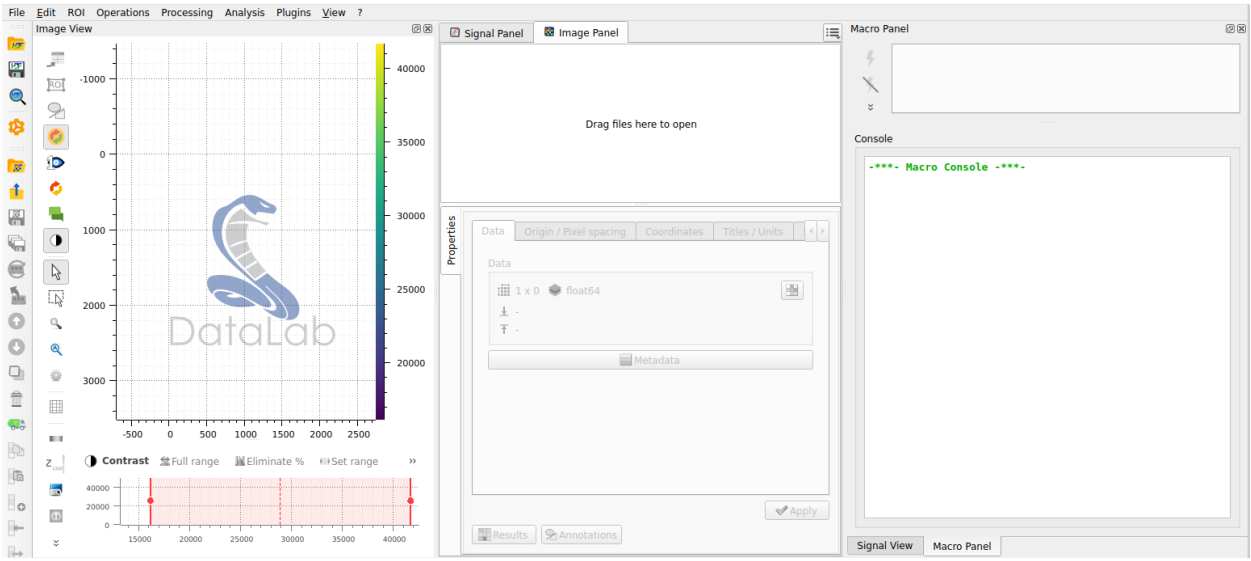


Fig. 95: The window layout of DataLab reorganized to have the "Image Panel" on the left and the "Macro Panel" on the right.

We can now generate a test image using the plugin that we installed before. To do so, we click on the "Plugins > Extract blobs (example) > Generate test image" menu. The function we are developing is quite slow, so we choose a limited size for the image (for example 512x512 pixels).

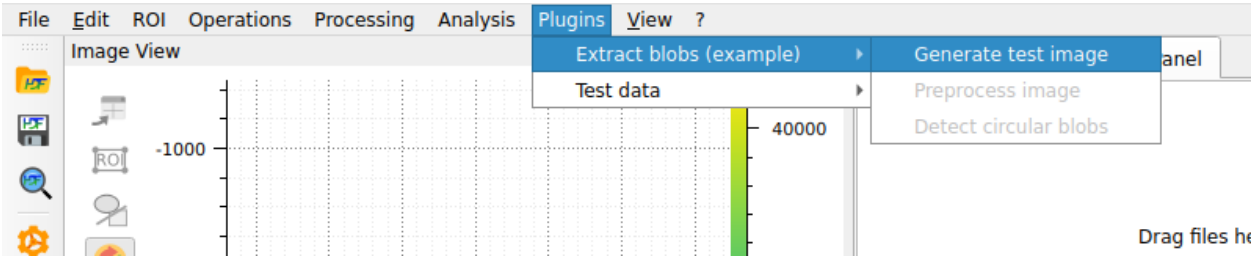


Fig. 96: We generate a new image using the "Plugins > Extract blobs (example) > Generate test image" menu.

The plugin generates a test image and adds it to the "Image Panel".

Create a macro-command

A macro is a code that can be executed directly inside DataLab to automate tasks. Macros are written in Python, and use the DataLab API to interact with DataLab.

In addition, macros have the following important features:

- Macros are part of DataLab's **workspace**, which means that they are saved and restored when exporting and importing to/from an HDF5 file.
- Macros are executed in a separate process, so we need to import the necessary modules and initialize the proxy to DataLab. The proxy is a special object that allows us to communicate with DataLab.
- As a consequence, **when defining a plugin or when controlling DataLab from an external IDE, we can use exactly the same code as in the macro-command**. This is a very important point, because it means that we can

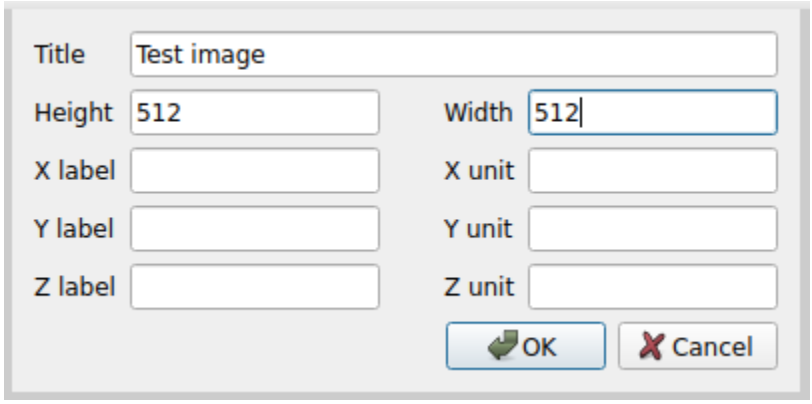


Fig. 97: We select the size for the image and click on “OK”.

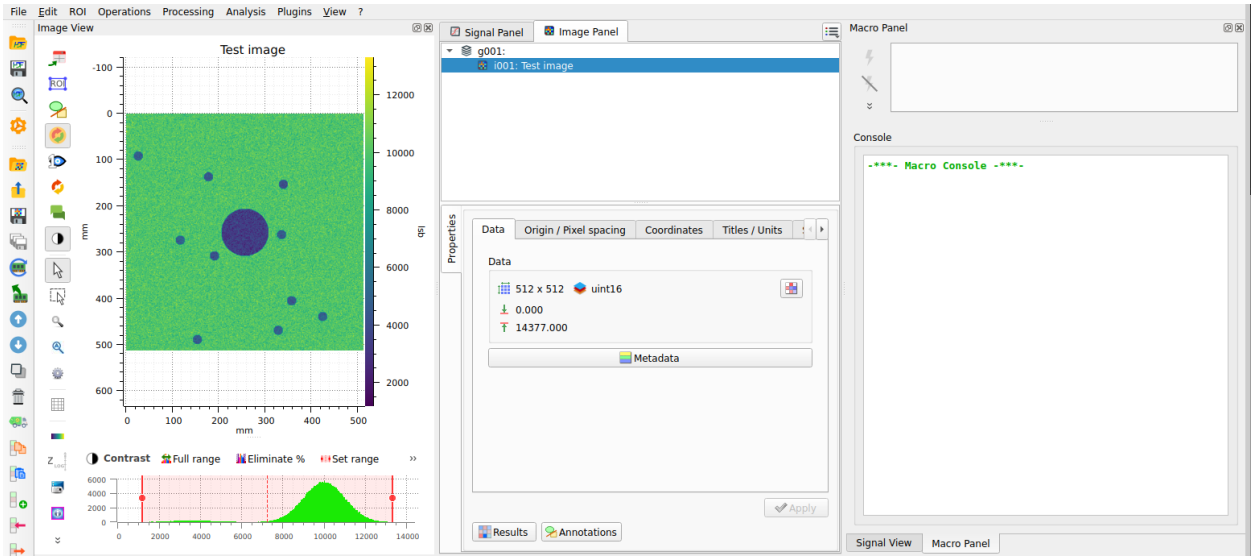


Fig. 98: The generated image in the “Image Panel”.

prototype our processing pipeline in DataLab, and then use the same code in a plugin or in an external IDE to develop it further.

Note: The macro-command is executed in DataLab's Python environment, so we can use the modules that are available in DataLab. However, we can also use our own modules, as long as they are installed in DataLab's Python environment or in a Python distribution that is compatible with DataLab's Python environment. A list of the modules that are available in your DataLab's Python environment is available in the "? > Installation and configuration > Installation configuration" dialog.

If your custom modules are not installed in DataLab's Python environment, and if they are compatible with DataLab's Python version, you can prepend the `sys.path` with the path to the Python distribution that contains your modules:

```
import sys
sys.path.insert(0, "/path/to/my/python/distribution")
```

This will allow you to import your modules in the macro-command and mix them with the modules that are available in DataLab.

Warning: If you use this method, make sure that your modules are compatible with DataLab's Python version. Otherwise, you will get errors when importing them.

To edit macros, we use the "Macro Panel" that is available in DataLab. This panel is separated in two parts: the upper part is the macro editor, where we can write and edit the macro code; the lower part is the console, where we can see the output of the macro. If the macro editor is too small to show all the buttons of the macro toolbar, some of them are hidden and can be accessed by clicking on the unfold button. Alternatively, you can resize the macro editor by dragging the splitter between the editor and the console.

Let's get back to our custom function. We can create a new macro-command that will apply the function to the current image. To do so, we open the "Macro Panel" and click on the "New macro" button.

DataLab creates a new macro-command which is not empty: it contains a sample code that shows how to create a new image and add it to the "Image Panel".

In this code, we can notice the presence of the `RemoteProxy` class that is used to communicate with DataLab. This class is part of the DataLab API, and is used to access DataLab objects and functions from a macro-command, a plugin, or an external IDE. You can find the documentation of the DataLab API in the *API* section.

We can remove this code and replace it with our own code:

```
# Import the necessary modules
import numpy as np
import scipy.ndimage as spi
from datalab.control.proxy import RemoteProxy

# Define our custom processing function
def weighted_average_denoise(values: np.ndarray) -> float:
    """Apply a custom denoising filter to an image.

    This filter averages the pixels in a 5x5 neighborhood, but gives less weight
    to pixels that significantly differ from the central pixel.
    """
    central_pixel = values[len(values) // 2]
    differences = np.abs(values - central_pixel)
```

(continues on next page)

(continued from previous page)

```
    weights = np.exp(-differences / np.mean(differences))
    return np.average(values, weights=weights)

# Initialize the proxy to DataLab
proxy = RemoteProxy()

# Switch to the "Image Panel" and get the current image
proxy.set_current_panel("image")
image = proxy.get_object()
if image is None:
    # We raise an explicit error if there is no image to process
    raise RuntimeError("No image to process!")

# Get a copy of the image data, and apply the function to it
data = np.array(image.data, copy=True)
data = spi.generic_filter(data, weighted_average_denoise, size=5)

# Add new image to the panel
proxy.add_image("My custom filtered data", data)
```

Now, let's execute the macro-command by clicking on the "Run macro" button:

- The macro-command is executed in a separate process, so we can continue to work in DataLab while the macro-command is running. And, if the macro-command takes too long to execute, we can stop it by clicking on the "Stop macro" button.
- During the execution of the macro-command, we can see the progress in the "Macro Panel" window: the process standard output is displayed in the "Console" below the macro editor. We can see the following messages:
  - `---[...]---[# ==> Running 'Untitled 01' macro...]`: the macro-command starts
  - `Connecting to DataLab XML-RPC server...OK [...]`: the proxy is connected to DataLab
  - `---[...]---[# <== 'Untitled 01' macro has finished]`: the macro-command ends

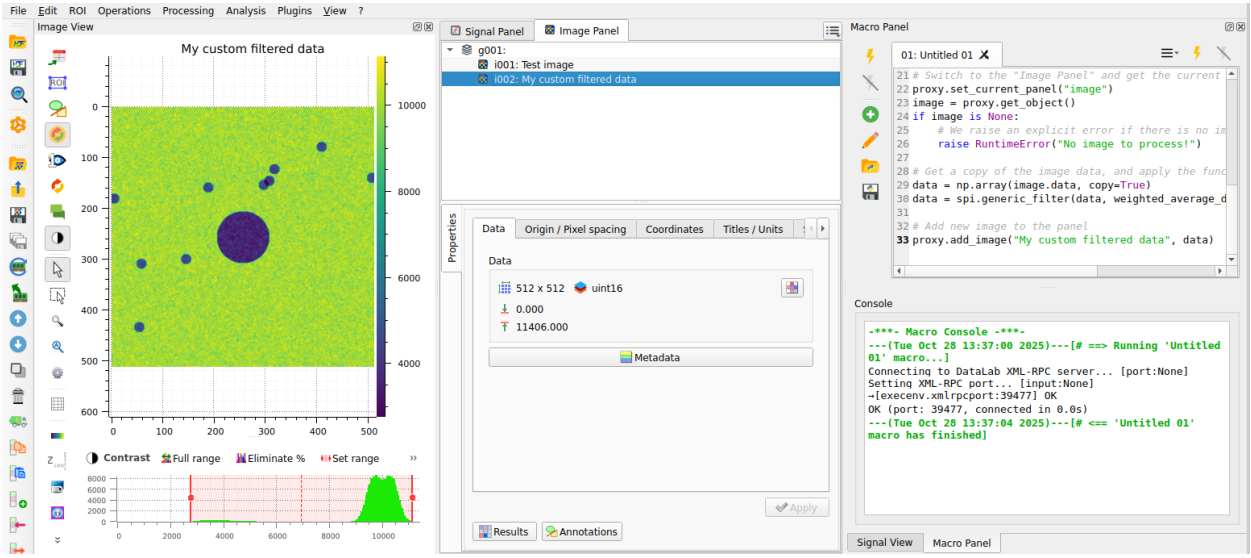


Fig. 99: When the macro-command has finished, we can see the new image in the "Image Panel". Our filter has been applied to the image, and we can see that the noise has been reduced.

Prototyping with an external IDE

Now that we have a working prototype of our processing pipeline, we can use the same code in an external IDE to develop it further.

For example, we can use the Spyder IDE to debug our code. To do so, we need to install Spyder but not necessarily in DataLab's Python environment (in the case of the stand-alone version of DataLab, it wouldn't be possible anyway).

The only requirement is to install a DataLab client in Spyder's Python environment:

- If you use the stand-alone version of DataLab or if you want or need to keep DataLab and Spyder in separate Python environments, you can install the Sigima (`sigima`) package that provides a simple remote client for DataLab. To install it, just run:

  pip install sigima

  Or you may also install the DataLab Python package (`datalab`) which includes the client (but also other modules, so we don't recommend this method if you don't need all DataLab's features in this Python environment):

  ```
  pip install datalab-platform
  ```

- If you use the DataLab Python package, you may run Spyder in the same Python environment as DataLab, so you don't need to install the client: it is already available in the main DataLab package (the `datalab` package).

Once the client is installed, we can start Spyder and create a new Python script:

```python
# -*- coding: utf-8 -*-
"""
Example of remote control of DataLab current session,
from a Python script running outside DataLab (e.g. in Spyder)

Created on Fri May 12 12:28:56 2023

@author: p.raybaut
"""

# %% Importing necessary modules

import numpy as np
import scipy.ndimage as spi
from sigima.client import SimpleRemoteProxy

# %% Connecting to DataLab current session

proxy = SimpleRemoteProxy()
proxy.connect()

# %% Executing commands in DataLab (...)


# Define our custom processing function
def weighted_average_denoise(data: np.ndarray) -> np.ndarray:
    """Apply a custom denoising filter to an image.

    This filter averages the pixels in a 5x5 neighborhood, but gives less weight
    to pixels that significantly differ from the central pixel.
```

(continues on next page)

(continued from previous page)

```
    """

    def filter_func(values: np.ndarray) -> float:
        """Filter function"""
        central_pixel = values[len(values) // 2]
        differences = np.abs(values - central_pixel)
        weights = np.exp(-differences / np.mean(differences))
        return np.average(values, weights=weights)

    return spi.generic_filter(data, filter_func, size=5)


# Switch to the "Image Panel" and get the current image
proxy.set_current_panel("image")
image = proxy.get_object()
if image is None:
    # We raise an explicit error if there is no image to process
    raise RuntimeError("No image to process!")

# Get a copy of the image data, and apply the function to it
data = np.array(image.data, copy=True)
data = weighted_average_denoise(data)

# Add new image to the panel
proxy.add_image("Filtered using Spyder", data)
```

We go back to DataLab and select the first image in the "Image Panel".

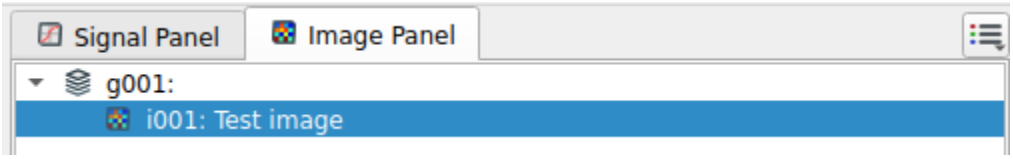


Fig. 100: The first image in the "Image Panel" selected.

Then, we execute the script in Spyder (or your preferred editor), step-by-step (using the defined cells), and we can see the result in DataLab.

We can see in DataLab that a new image has been added to the "Image Panel". This image is the result of the execution of the script in Spyder.

Here we have used the script without any modification, but we could have modified it to test new ideas, and then use the modified script in DataLab.

There is another useful application of having an external script interfaced with DataLab: you can imagine letting your external script acquire the data and add it to DataLab automatically, for example in an acquisition loop. In this way you can improve your measurement workflow with the possibility to visualize and analyze data directly after their acquisition.

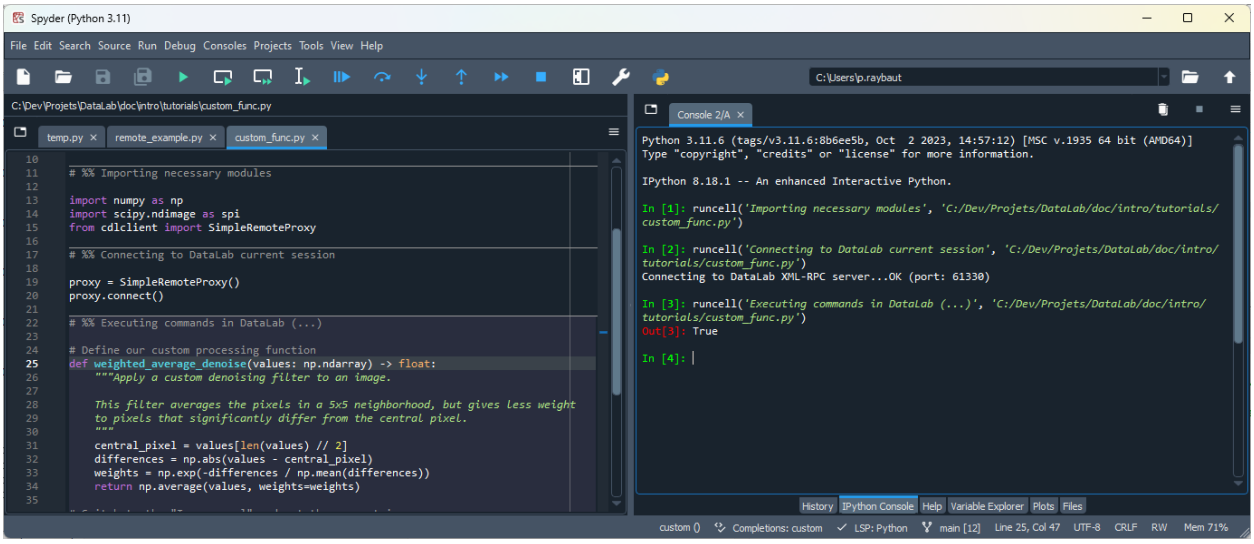


Fig. 101: The Spyder editor environment with the custom processing script.

Prototyping with a Jupyter notebook

We can also use a Jupyter notebook to prototype our processing pipeline. To do so, we need to install Jupyter but not necessarily in DataLab's Python environment (in the case of the stand-alone version of DataLab, it wouldn't be possible anyway).

The procedure is the same as for Spyder or any other IDE like Visual Studio Code, for example.

Creating a plugin

Now that we have a working prototype of our processing pipeline, we can choose to create a plugin to integrate it in DataLab's GUI. To do so, we need to create a new Python module that will contain the plugin code. We can use the same code as in the macro-command, but we need to make some changes.

See also:

The plugin system is described in the *Plugins* section.

Apart from integrating the feature to DataLab's GUI which is more convenient for the user, the advantage of creating a plugin is that we can take benefit of the DataLab infrastructure if we encapsulate our processing function in a certain way (see below):

- Our function will be executed in a separate process, so we can interrupt it if it takes too long to execute.
- Warnings and errors will be handled by DataLab, so we don't need to handle them ourselves.

The most significant change we need to make to our code is that we need to define a function that will be operating on DataLab's native image objects (`sigima.objects.ImageObj`), instead of operating on NumPy arrays. So we need to find a way to call our custom function `weighted_average_denoise` with a `sigima.objects.ImageObj` as input and output. To avoid writing a lot of boilerplate code, we can use the function wrapper provided by DataLab: `sigima.proc.image.Wrap1to1Func`.

Besides we need to define a class that describes our plugin, which must inherit from `datalab.plugins.PluginBase` and name the Python script that contains the plugin code with a name that starts with `datalab_` (e.g., `datalab_custom_func.py`), so that DataLab can discover it at startup.

DataLab custom function example

This is part of the DataLab's custom function tutorial which aims at illustrating the extensibility of DataLab (macros, plugins, and control from an external IDE or a Jupyter notebook).

The only requirement is to install the *DataLab Simple Client* package (using `pip install cdlclient`, for example).

[2]:
```
import numpy as np
import scipy.ndimage as spi
from cdlclient import SimpleRemoteProxy

# Define our custom processing function
def weighted_average_denoise(values: np.ndarray) -> float:
    """Apply a custom denoising filter to an image.

    This filter averages the pixels in a 5x5 neighborhood, but gives less weight
    to pixels that significantly differ from the central pixel.
    """
    central_pixel = values[len(values) // 2]
    differences = np.abs(values - central_pixel)
    weights = np.exp(-differences / np.mean(differences))
    return np.average(values, weights=weights)
```

Connecting to DataLab current session:

[3]:
```
proxy = SimpleRemoteProxy()
proxy.connect()
```

```
Connecting to DataLab XML-RPC server...OK (port: 61330)
```

Switch to the "Image panel" and get the current image

[5]:
```
proxy.set_current_panel("image")
image = proxy.get_object()
if image is None:
    # We raise an explicit error if there is no image to process
    raise RuntimeError("No image to process!")
```

Get a copy of the image data, apply the function to it, and add new image to the panel

[6]:
```
data = np.array(image.data, copy=True)
data = spi.generic_filter(data, weighted_average_denoise, size=5)
proxy.add_image("Filtered using a Jupyter notebook", data)
```

Fig. 102: A code interfaced with datalab written in a Jupyter notebook.

Moreover, inside the plugin code, we want to add an entry in the "Plugins" menu, so that the user can access our plugin from the GUI.

Here is the plugin code:

```python
# -*- coding: utf-8 -*-

"""
Custom denoising filter plugin
==============================

This is a simple example of a DataLab image processing plugin.

It is part of the DataLab custom function tutorial.

.. note::

    This plugin is not installed by default. To install it, copy this file to
    your DataLab plugins directory (see `DataLab documentation
    <https://datalab-platform.com/en/features/advanced/plugins.html>`_).
"""

import numpy as np
import scipy.ndimage as spi
import sigima.proc.image as sipi

import datalab.plugins


def weighted_average_denoise(data: np.ndarray) -> np.ndarray:
    """Apply a custom denoising filter to an image.

    This filter averages the pixels in a 5x5 neighborhood, but gives less weight
    to pixels that significantly differ from the central pixel.
    """

    def filter_func(values: np.ndarray) -> float:
        """Filter function"""
        central_pixel = values[len(values) // 2]
        differences = np.abs(values - central_pixel)
        weights = np.exp(-differences / np.mean(differences))
        return np.average(values, weights=weights)

    return spi.generic_filter(data, filter_func, size=5)


class CustomFilters(datalab.plugins.PluginBase):
    """DataLab Custom Filters Plugin"""

    PLUGIN_INFO = datalab.plugins.PluginInfo(
        name="My custom filters",
        version="1.0.0",
        description="This is an example plugin",
    )
```

(continues on next page)

(continued from previous page)

```
    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        proc = self.imagepanel.processor
        with acth.new_menu(self.PLUGIN_INFO.name):
            for name, func in (("Weighted average denoise", weighted_average_denoise),):
                # Wrap function to handle ``ImageObj`` objects instead of NumPy arrays
                wrapped_func = sipi.Wrap1to1Func(func)
                acth.new_action(
                    name,
                    triggered=lambda: proc.compute_1_to_1(wrapped_func, title=name),
                )
```

To test it, we have to add the plugin script to one of the plugin directories that are discovered by DataLab at startup: you can find it under "Files > Settings" (see the *Plugins* section for more details, or the *Detecting blobs on an image* for an example). We then restart DataLab and we can see that the plugin has been loaded.

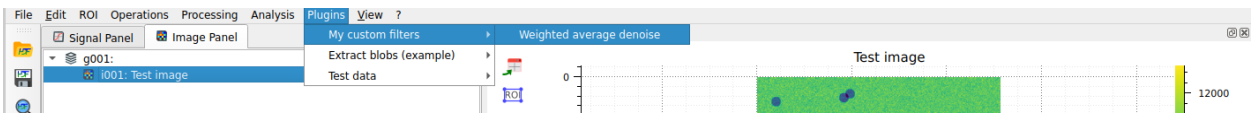


Fig. 103: At the restart we can see that the plugin has been loaded.

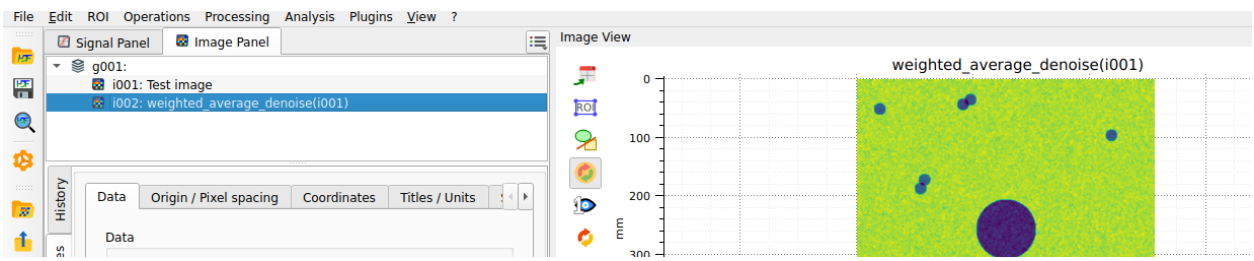
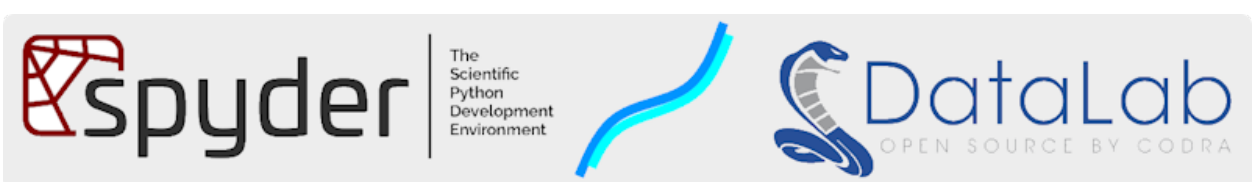


Fig. 104: We generate again our test image using (see the first steps of the tutorial), and we process it using the plugin: "Plugins > My custom filters > Weighted average denoise".

DataLab and Spyder: a perfect match

This tutorial demonstrates how to use Spyder to work with DataLab through a practical example, using sample algorithms and data that represent a hypothetical research or technical workflow. The goal is to illustrate how to use DataLab to test your algorithms with data and debug them when necessary.

The example is straightforward, yet it demonstrates the essential concepts of working with both DataLab *and* Spyder.



Note: DataLab and Spyder are **complementary** tools. While Spyder is a powerful development environment with

interactive scientific computing capabilities, DataLab is a versatile data analysis platform that can perform a wide range of tasks, from simple data visualization to complex data analysis and processing. In other words, Spyder is a **development** tool, while DataLab is a **data analysis** tool. You can use Spyder to develop algorithms and then use DataLab to analyze data with those algorithms.

Basic concepts

In the context of your research or technical work, we assume that you are developing software to process data (signals or images). This software may be a stand-alone application, a library for use in other applications, or even a simple script to run from the command line. In any case, you will need to follow a development process that typically includes the following steps:

0. Prototype the algorithm in a development environment such as Spyder.
1. Develop the algorithm in a development environment such as Spyder.
2. Test the algorithm with data.
3. Debug the algorithm if necessary.
4. Repeat steps 2 and 3 until the algorithm works as expected.
5. Use the algorithm in your application.

Note: DataLab can help you with step 0 because it provides all the processing primitives you need to prototype your algorithm: you can load data, visualize it, and perform basic processing operations. We won't cover this step in the following sections since the DataLab documentation already provides extensive information about it.

In this tutorial, we will see how to use DataLab to perform steps 2 and 3. We assume that you have already prototyped (preferably in DataLab!) and developed your algorithm in Spyder. Now, you want to test it with data, but without leaving Spyder because you may need to modify your algorithm and re-test it. Besides, your workflow is already set up in Spyder and you don't want to change it.

Note: This tutorial assumes that you have already installed DataLab and started it. If you haven't done so yet, please refer to the *Installation* section of the documentation.

Additionally, we assume that you have already installed Spyder and started it. If you haven't done so yet, please refer to the Spyder documentation. **Note that you don't need to install DataLab in the same environment as Spyder:** that's the whole point of DataLab—it is a stand-alone application that can be used from any environment. For this tutorial, you only need to install the Sigima package (`pip install sigima`) in the same environment as Spyder.

Testing your algorithm with DataLab

Let's assume that you have developed algorithms in the `my_work` module of your project. You have already prototyped them in DataLab and developed them in Spyder by writing functions that take data as input and return processed data as output. Now, you want to test these algorithms with data.

To test these algorithms, you have written two functions in the `my_work` module:

- `test_my_1d_algorithm`: this function returns 1D data that will allow you to validate your first algorithm, which processes 1D data.

- `test_my_2d_algorithm`: this function returns 2D data that will allow you to validate your second algorithm, which processes 2D data.

You can now use DataLab to visualize the data returned by these functions directly from Spyder:

- First, start both DataLab and Spyder.
- Remember that DataLab is a stand-alone application that can be used from any environment, so you don't need to install it in the same environment as Spyder because the connection between these two applications is established through a communication protocol.

Here is how to do it:

```
# %% Connecting to DataLab current session

from sigima.client import SimpleRemoteProxy

proxy = SimpleRemoteProxy()
proxy.connect()

# %% Visualizing 1D data from my work

from my_work import test_my_1d_algorithm

x, y = test_my_1d_algorithm()  # Here is all my research/technical work!
proxy.add_signal("My 1D result data", x, y)  # Let's visualize it in DataLab
proxy.compute_wiener()  # Denoise the signal using the Wiener filter


# %% Visualizing 2D data from my work

from my_work import test_my_2d_algorithm

z = test_my_2d_algorithm()[1]  # Here is all my research/technical work!
proxy.add_image("My 2D result data", z)  # Let's visualize it in DataLab
```

If we execute the first two cells, we will see the following output in the Spyder console:

```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.15.0 -- An enhanced Interactive Python.

In [1]: runcell('Connecting to DataLab current session', 'my_work_test_with_datalab.py')
Connecting to DataLab XML-RPC server...OK (port: 54577)

In [2]: runcell('Visualizing 1D data from my work', 'my_work_test_with_datalab.py')
Out[2]: True
```

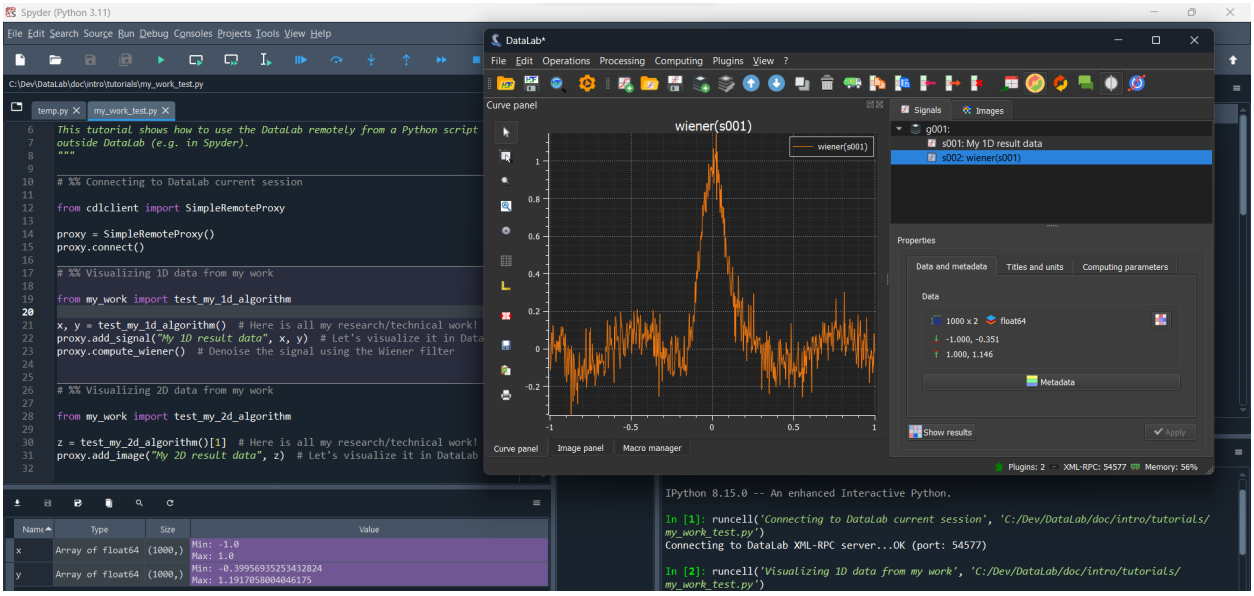


Fig. 105: In this screenshot, we can see the result of evaluating the first two cells: the first cell connects to DataLab, and the second cell visualizes the 1D data returned by the `test_my_1d_algorithm` function.

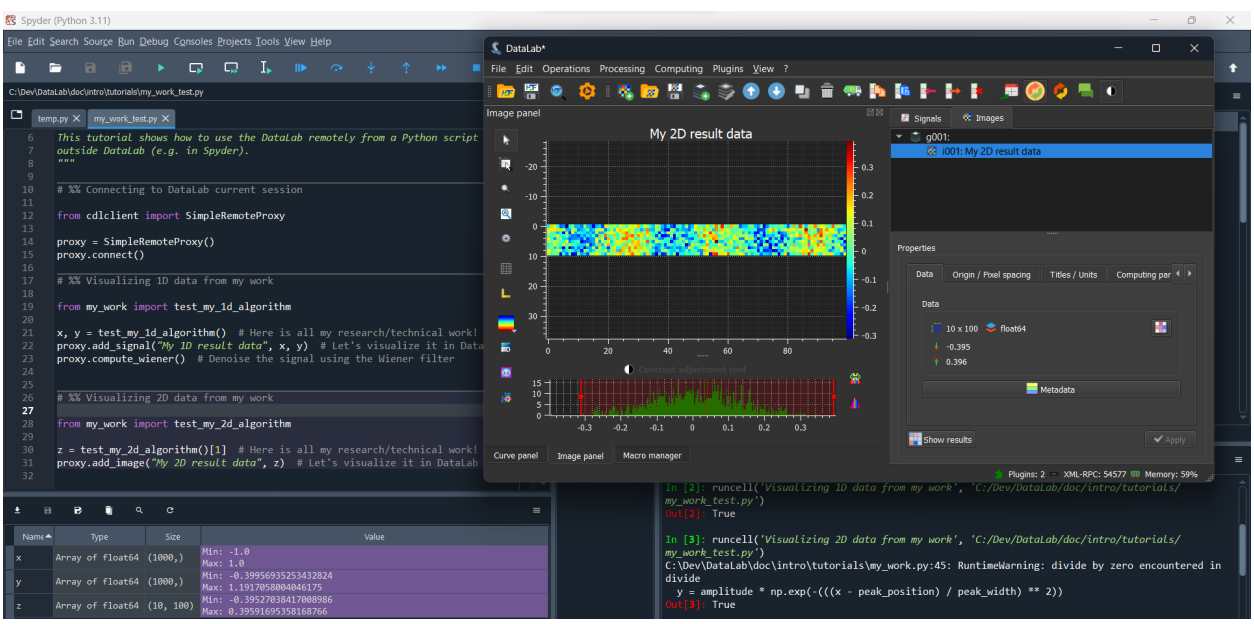


Fig. 106: In this screenshot, we can see the result of evaluating the third cell: the `test_my_2d_algorithm` function returns a 2D array, which we can visualize directly in DataLab.

Debugging your algorithm with DataLab

Now that you have tested your algorithms with data, you may want to debug them if necessary. To do so, you can combine the Spyder debugging capabilities with DataLab.

Here is the code of the sample algorithm that we want to debug, in which we have introduced an optional `debug_with_datalab` parameter that—if set to `True`— will create a proxy object allowing step-by-step visualization of the data in DataLab:

```
def generate_2d_data(
    num_lines: int = 10,
    noise_std: float = 0.1,
    x_min: float = -1,
    x_max: float = 1,
    num_points: int = 100,
    debug_with_datalab: bool = False,
) -> tuple[np.ndarray, np.ndarray]:
    """Generate 2D data that can be used in the tutorials to represent some results.

    Args:
        num_lines: Number of lines in the generated data, by default 10.
        noise_std: Standard deviation of the noise, by default 0.1.
        x_min: Minimum value of the x axis, by default -1.
        x_max: Maximum value of the x axis, by default 1.
        num_points: Number of points in the generated data, by default 100.
        debug_with_datalab: Whether to use the DataLab to debug the function,
         by default False.

    Returns:
        Generated data (x, y; where y is a 2D array and x is a 1D array).
    """
    proxy = None
    if debug_with_datalab:
        proxy = SimpleRemoteProxy()
        proxy.connect()
    z = np.zeros((num_lines, num_points))
    for i in range(num_lines):
        amplitude = 0.1 * i**2
        peak_position = 0.5 * i**2
        peak_width = 0.1 * i**2
        x, y = generate_1d_data(
            amplitude,
            peak_position,
            peak_width,
            relaxation_oscillations=True,
            noise=True,
            noise_std=noise_std,
            x_min=x_min,
            x_max=x_max,
            num_points=num_points,
        )
        z[i] = y
        if proxy is not None:
            proxy.add_signal(f"Line {i}", x, y)
```

(continues on next page)

(continued from previous page)

```
    return x, z
```

The corresponding `test_my_2d_algorithm` function also has an optional `debug_with_datalab` parameter that is simply passed to the `generate_2d_data` function.

Now, we can use Spyder to debug the `test_my_2d_algorithm` function:

```
# %% Debugging my work with DataLab

from my_work import generate_2d_data

x, z = generate_2d_data(debug_with_datalab=True)
```

In this simple example, the algorithm iterates 10 times and generates a 1D array at each iteration. Each 1D array is then stacked into a 2D array that is returned by the `generate_2d_data` function. With the `debug_with_datalab` parameter set to `True`, we can visualize each 1D array in DataLab: that way, we can verify that the algorithm is working as expected.

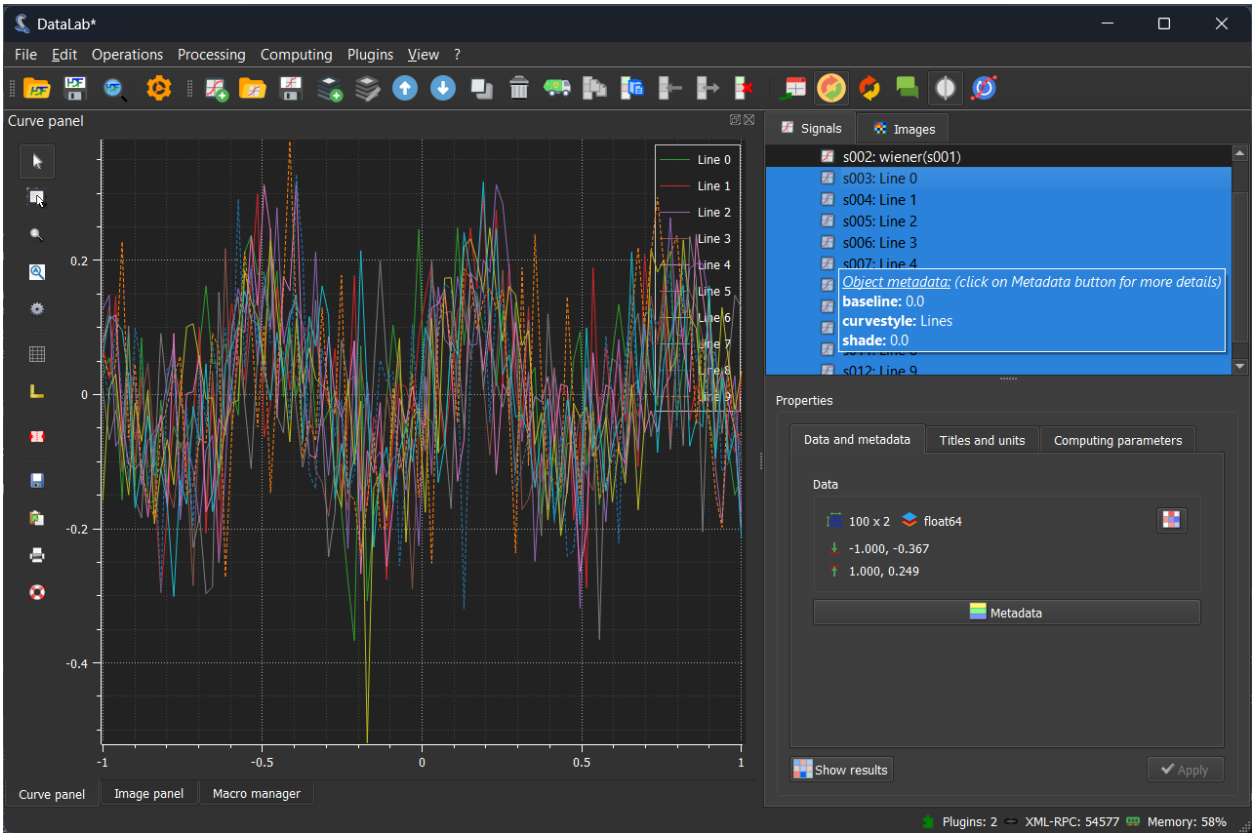


Fig. 107: In this screenshot, we can see the result of evaluating the first cell: the `test_my_2d_algorithm` function is called with the `debug_with_datalab` parameter set to `True`: 10 1D arrays are generated and visualized in DataLab.

Note: If we had executed the script using the Spyder debugger and set a breakpoint in the `generate_2d_data` function, we would have seen the generated 1D arrays in DataLab at each iteration. Since DataLab runs in a separate process, we would have been able to interact with the data in DataLab while the algorithm is paused in Spyder.

CHAPTER

TWO

FEATURES

The following sections describe the features of DataLab, the open-source scientific data analysis and visualization platform.

Note: Before jumping into the details of other parts of the documentation, it is recommended to start with the *Overview* page, which describes the basic concepts of DataLab.

See also:

For a synthetic overview of the features of DataLab, please refer to the *Key features* page.

2.1 Overview & Common features

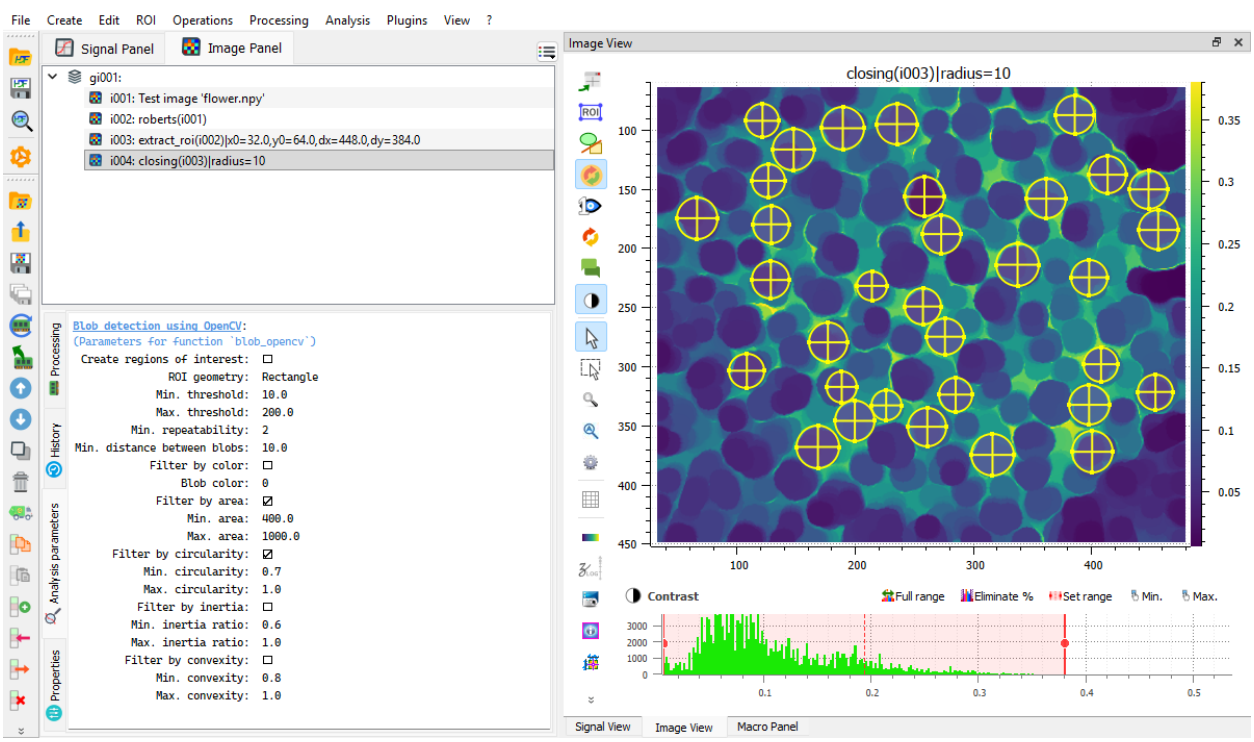


Fig. 1: DataLab main window

2.1.1 Overview

Basic concepts

Working with DataLab is very easy. The user interface is intuitive and self-explanatory. The main window is divided into two main areas:

- The left area shows the list of data sets which are currently loaded in DataLab, distibuted over two tabs: **Signals** and **Images**. The user can switch between the two tabs by clicking on the corresponding tab: this switches the main window to the corresponding panel, as well as the menu and toolbar contents. Below the list of data sets, a **Properties** view shows information about the currently selected data set.
- The right area shows the visualization of the currently selected data set. The visualization is updated automatically when the user selects a new data set in the list of data sets.

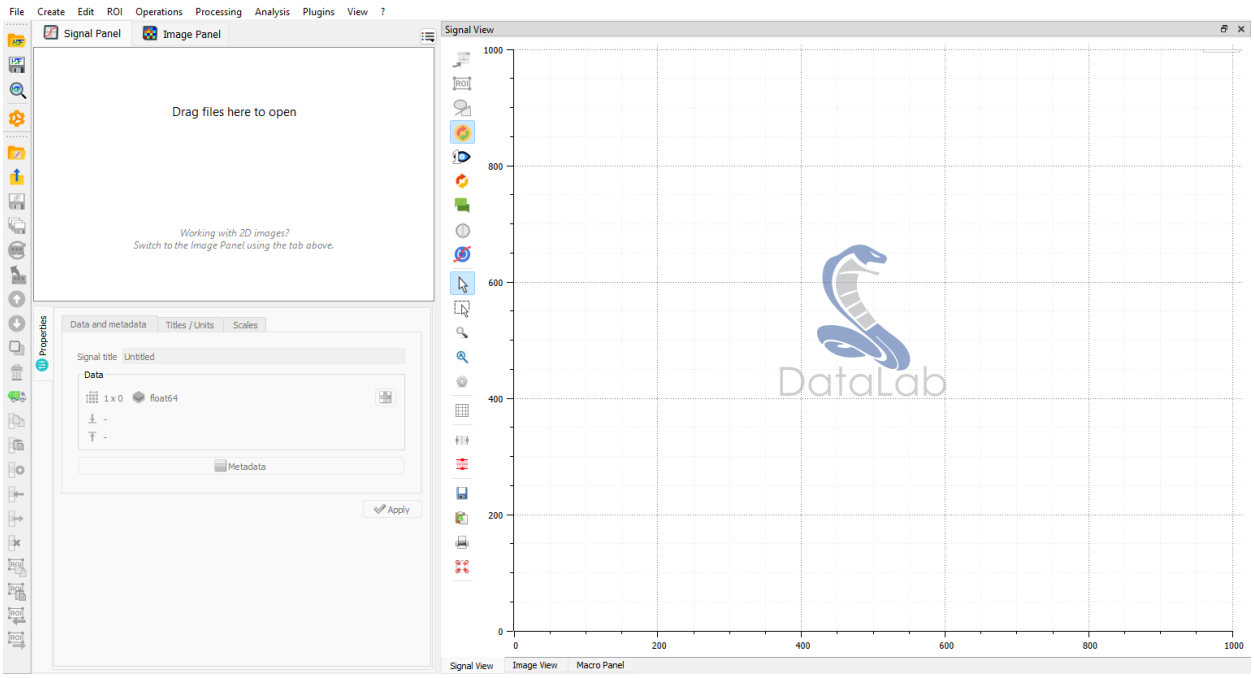


Fig. 2: DataLab main window, at startup.

Internal data model and workspace

DataLab has its own internal data model, in which data sets are organized around a tree structure. Each panel in the main window corresponds to a branch of the tree. Each data set shown in the panels corresponds to a leaf of the tree. Inside the data set, the data is organized in an object-oriented way, with a set of attributes and methods. The data model is described in more details in the API section (see `sigima.objects`).

For each data set (1D signal or 2D image), not only the data itself is stored, but also a set of metadata, which describes the data or the way it has to be displayed. The metadata is stored in a dictionary, which is accessible through the `metadata` attribute of the data set (and may also be browsed in the **Properties** view, with the **Metadata** button).

The DataLab **Workspace** is defined as the collection of all data sets which are currently loaded in DataLab, in both the **Signals** and **Images** panels.

Loading and saving the workspace

The following actions are available to manage the workspace from the **File** menu:

- **Open HDF5 file**: load a workspace from an HDF5 file.
- **Save to HDF5 file**: save the current workspace to an HDF5 file.
- **Browse HDF5 file**: open the *HDF5 Browser* to explore the content of an HDF5 file and import data sets into the workspace.

Note: Data sets may also be saved or loaded individually, using data formats such as *.txt* or *.npy* for 1D signals (see *Open signal* for the list of supported formats), , or *.tiff* or *.dcm* for 2D images (see *Open image* for the list of supported formats).

Interactive object creation and processing

DataLab provides an interactive workflow for creating objects and adjusting processing parameters, allowing you to fine-tune results without creating multiple objects.

Interactive object creation

When creating a new signal or image using the creation functions (e.g., Gaussian signal, 2D peak image, etc.), DataLab stores the creation parameters in the object's metadata. This enables interactive parameter adjustment after creation:

1. Create a signal or image using **Operations > Create** menu
2. Select the created object in the list
3. A **Creation** tab appears in the Properties panel (bottom-left)
4. Modify any creation parameter (amplitude, frequency, size, etc.)
5. Click **Apply** to regenerate the object with new parameters

The object is updated in-place, preserving any subsequent processing or analysis results. This is particularly useful for:

- Exploring different parameter values without cluttering the workspace
- Fine-tuning object characteristics while observing the results
- Educational purposes to demonstrate parameter effects

Note: Interactive creation is only available for objects created with parameter classes (not for imported data from files).

Interactive 1-to-1 processing

When applying a 1-to-1 processing operation that has configurable parameters (e.g., Gaussian filter, threshold, morphological operations), DataLab stores the processing metadata, enabling parameter adjustment and re-processing:

1. Apply a processing operation with parameters (e.g., **Processing** > **Filtering** > **Gaussian filter**)
2. The result object contains processing metadata (parameters, source object, function name)
3. Select the processed object in the list
4. A **Processing** tab appears in the Properties panel
5. Modify processing parameters (e.g., filter sigma value)
6. Click **Apply** to re-process with updated parameters

The processed object is updated in-place with the new results. This workflow is ideal for:

- Iteratively tuning filter parameters while observing results in real-time
- Adjusting threshold values without creating multiple intermediate objects
- Experimenting with different morphological structure element sizes
- Educational demonstrations of parameter effects on processing results

---

Note: Processing metadata requirements:

- Only works for 1-to-1 processing functions with parameter classes
- Processing functions without parameters (e.g., absolute value, inverse) work as before
- Source object must still exist for re-processing (error shown if deleted)

Not supported for:

- 2-to-1 processing (operations combining two objects)
- n-to-1 processing (operations aggregating multiple objects)
- These patterns don't benefit significantly from interactive parameter adjustment

---

Example workflow

Here's a typical workflow using interactive processing:

1. **Create a test signal**: Operations > Create > Gaussian signal
   - Initial parameters: amplitude=1.0, mu=50, sigma=10
2. **Adjust creation parameters**: In the Creation tab, change sigma to 20, click Apply
   - Signal is regenerated with new width
3. **Apply Gaussian filter**: Processing > Filtering > Gaussian filter
   - Initial sigma=2.0
4. **Fine-tune filtering**: In the Processing tab, try sigma=1.0, then sigma=5.0
   - Each Apply updates the filtered result
   - Compare different smoothing levels without creating multiple objects

This interactive approach provides immediate visual feedback and keeps your workspace organized by avoiding proliferation of test objects with slightly different parameters.

2.1.2 Settings

DataLab provides a comprehensive settings dialog to customize the application behavior, visualization defaults, and I/O operations. The settings are organized into five tabs: General, Processing, Visualization, I/O, and Console.

General

The General settings tab contains main window and general feature settings:

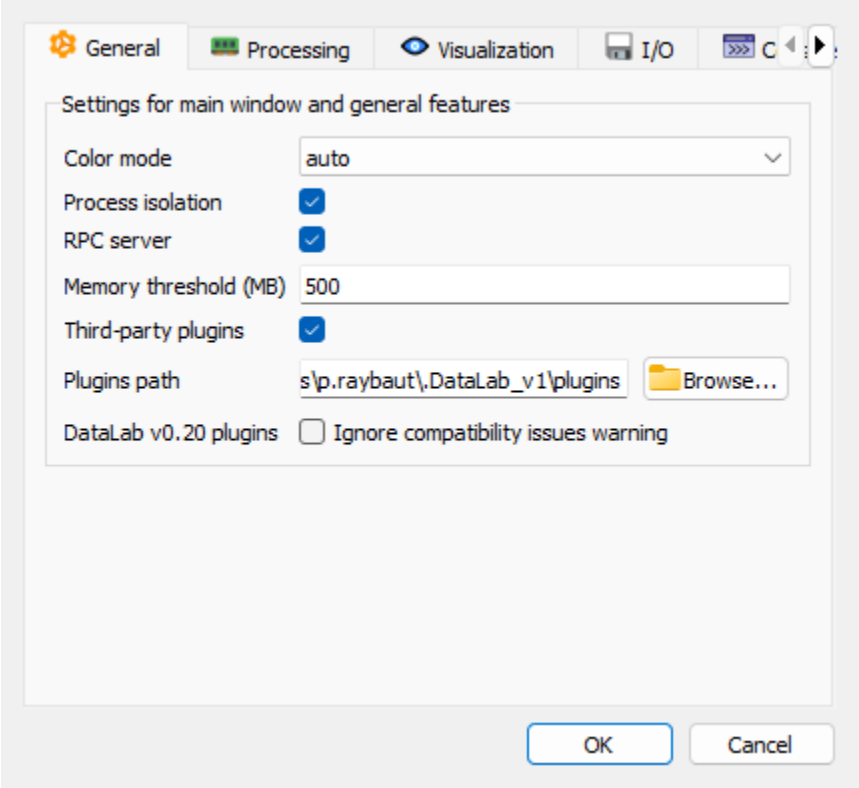


Fig. 3: General settings tab

Color mode
Choose the color mode for the application interface (e.g., light, dark, or auto).

Process isolation
When enabled, each computation runs in a separate process, preventing the application from freezing during long computations. This is the recommended setting for better responsiveness.

RPC server
Enable the RPC (Remote Procedure Call) server to communicate with external applications, such as your own scripts running in Spyder, Jupyter, or other software. This allows programmatic control of DataLab.

Memory threshold
Set a threshold (in MB) below which a warning is displayed before loading new data. This helps prevent out-of-memory errors when working with large datasets. Set to 0 to disable the warning.

Third-party plugins
Enable or disable third-party plugins at startup.

Plugins path
Specify the directory path where DataLab should look for third-party plugins. DataLab will also discover plugins in your PYTHONPATH.

Processing

The Processing settings tab controls computation behavior and default parameters:

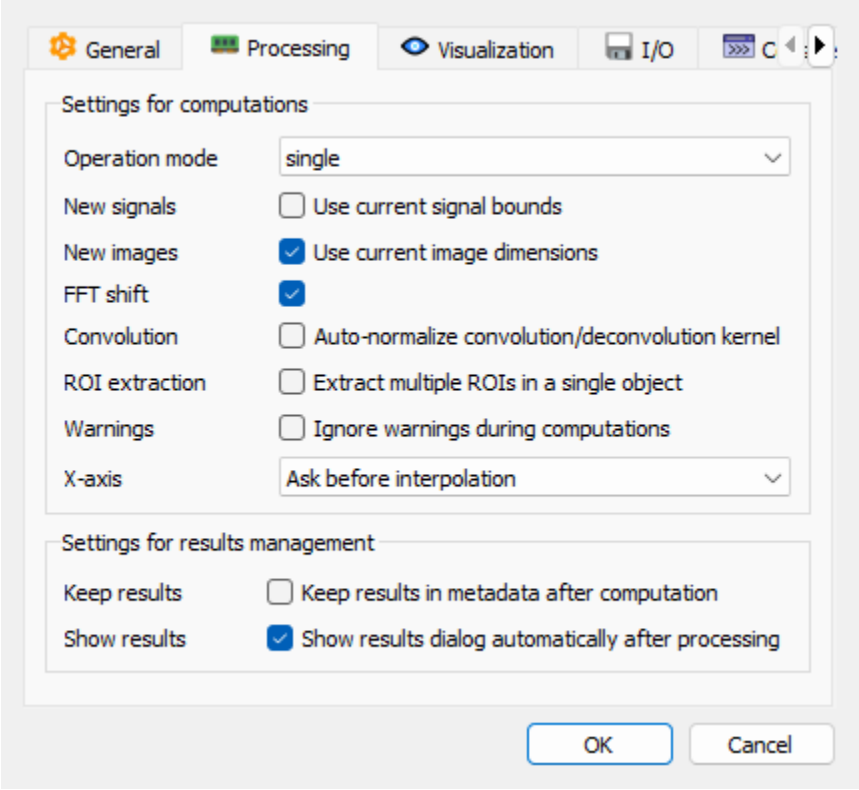


Fig. 4: Processing settings tab

Operation mode
Choose the operation mode for computations taking N inputs:

- **Single**: single operand mode
- **Pairwise**: pairwise operation mode

Note: These operation modes determine how DataLab handles computations involving multiple objects. They apply to two types of operations:

- **N→1 operations**: Combine N (2) objects into 1 output (e.g. sum, average)
- **N+1→N operations**: Apply an operation between N (1) objects and 1 operand to produce N outputs (e.g. difference, division)

Single operand mode (default): Operations are applied independently within each group.

- For **N→1 operations**: All objects in each group are combined into one result per group. Example with groups G1={A, B} and G2={C, D}, sum operation:
  - Result: (A,B) and (C,D) (one per group)
- For **N+1→N operations**: Each selected object is combined with a single reference operand. Example with groups G1={A, B} and G2={C, D}, difference with reference R:
  - In G1: A-R, B-R
  - In G2: C-R, D-R

Pairwise operation mode: Objects from different groups are combined at matching positions (all groups must have the same number of objects).

- For **N→1 operations**: Objects at the same position in each group are combined. Example with groups G1={A, B} and G2={C, D}, sum operation:
  - Result: A+C and B+D (pairing by position)
- For **N+1→N operations**: Objects at matching positions are combined pairwise. Example with groups G1={A, B}, G2={C, D}, difference with group G3={E, F}:
  - New group 1: A-E, B-F
  - New group 2: C-E, D-F

Use signal bounds for new signals
When enabled, the xmin and xmax values for new signals are initialized from the current signal's bounds. When disabled, default values are used.

Use image dimensions for new images
When enabled, the width and height values for new images are initialized from the current image's dimensions. When disabled, default values are used.

FFT shift
Enable FFT shift to center the zero-frequency component in the frequency spectrum for easier visualization and analysis.

Extract multiple ROIs in a single object
When enabled, multiple ROIs (Regions of Interest) are extracted into a single object. When disabled, each ROI is extracted into a separate object.

Ignore warnings
Suppress warning messages during computations.

X-array compatibility behavior
Choose the behavior when X arrays are incompatible in multi-signal computations:

- **Ask**: display a confirmation dialog (default)
- **Interpolate**: automatically interpolate signals

Result management

Keep results in metadata after computation
When enabled, results from previous analyses are kept in the object's metadata after computation. When disabled, results are removed. This option is disabled by default to avoid confusion from outdated results.

Show results dialog automatically after processing
When enabled, the results dialog is shown automatically after each processing operation producing results. When disabled, the results dialog is not shown automatically but results can still be viewed using the dedicated button or menu option.

Visualization

The Visualization settings tab controls how data is displayed. This tab is organized into four sub-tabs: Common, Signals, Images, and Results.

Common settings

The Common sub-tab contains settings that apply to all visualizations:

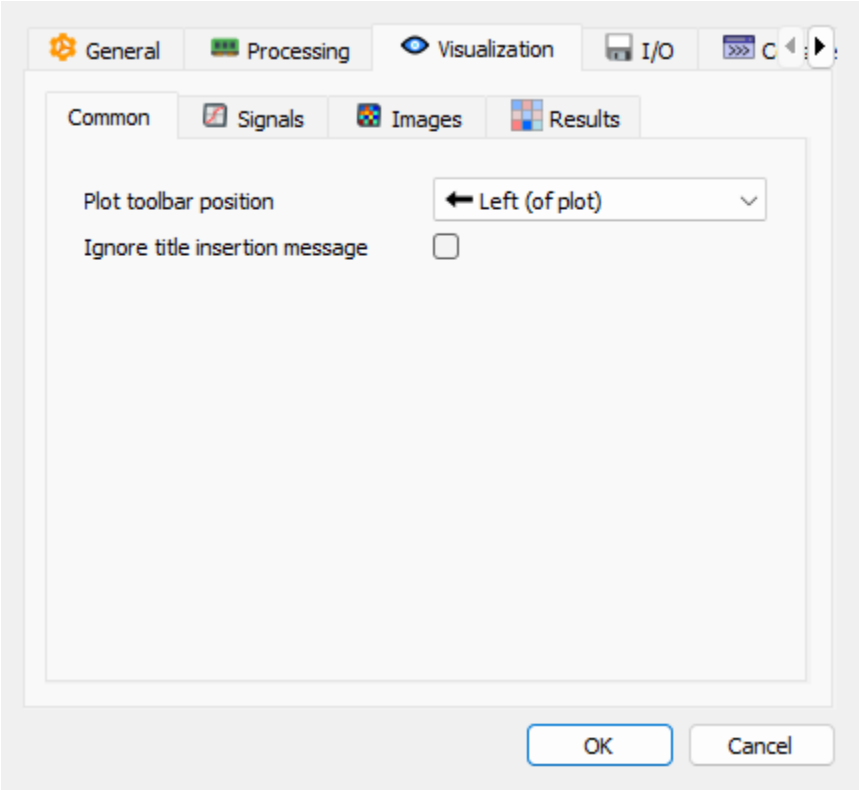


Fig. 5: Visualization settings - Common sub-tab

Plot toolbar position
Choose where to position the plot toolbar (top, bottom, left, or right of the plot).

Ignore title insertion message
Suppress the information message when inserting an object title as an annotation label.

Signals

The Signals sub-tab contains settings specific to signal visualizations:

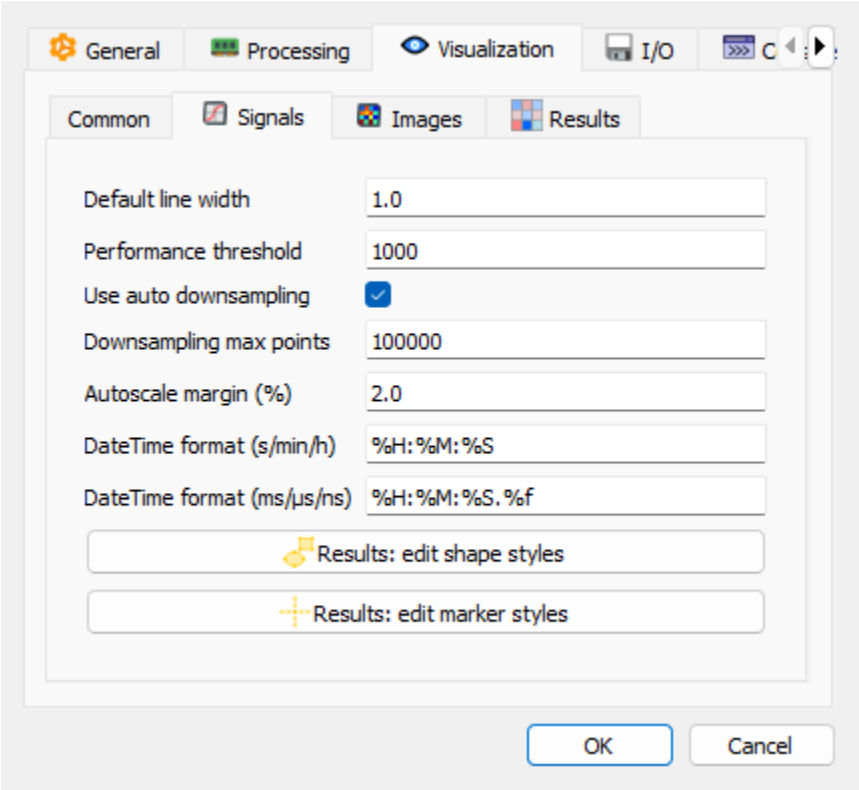


Fig. 6: Visualization settings - Signals sub-tab

Default line width
Default line width for curves representing signals. This setting affects all signal visualizations unless overridden individually. Note: for signals exceeding the line width performance threshold (see below), the line width is automatically clamped to 1.0 for optimal rendering performance.

Line width performance threshold
For signals with more than this number of points (default: 1,000), line width is automatically limited to 1.0 for performance reasons. This prevents the ~10x rendering slowdown caused by Qt's raster engine when drawing thick lines (width > 1.0) on large datasets. For smaller signals, the configured default line width applies normally. This optimization is transparent and requires no user intervention.

Use auto downsampling
Enable automatic downsampling for large signals to improve performance and visualization clarity.

Downsampling max points
Maximum number of points to display when downsampling is enabled (default: 10,000).

Autoscale margin
Percentage of margin to add around data when auto-scaling signal plots. A value of 0.2% adds a small margin for better visualization. Set to 0% for no margin (exact data bounds).

DateTime format (s/min/h)
Format string for datetime X-axis labels when using standard time units (s, min, h). Uses Python's strftime format codes (e.g., %H:%M:%S for hours:minutes:seconds).

DateTime format (ms/s/ns)
Format string for datetime X-axis labels when using sub-second time units (ms, s, ns). Uses Python's strftime format codes (e.g., %H:%M:%S.%f for hours:minutes:seconds.microseconds).

Results: edit shape styles
Click this button to configure the visual style for annotation shapes (rectangles, circles, segments, etc.) displayed on signal plots. This includes:

- Line style, color, and width
- Fill pattern, color, and transparency
- Symbol shape, size, and colors

These settings apply to all result shapes drawn on signal plots (e.g., peak markers, FWHM indicators, feature detection results).

Results: edit marker styles
Click this button to configure the visual style for cursor markers on signal plots. This includes:

- Line style, color, and width
- Symbol appearance
- Text label formatting and positioning
- Background transparency

These settings apply to cursor-type markers used in signal analysis results.

Images

The Images sub-tab contains settings specific to image visualizations:

Lock aspect ratio to 1:1
When enabled, the aspect ratio of images is locked to 1:1. When disabled, the aspect ratio is determined by the physical pixel size (default and recommended setting).

Eliminate outliers
Percentage of the highest and lowest values to eliminate from the image histogram. Recommended values are below 1%.

Autoscale margin
Percentage of margin to add around data when auto-scaling image plots. A value of 0.2% adds a small margin for better visualization. Set to 0% for no margin (exact data bounds).

Default image visualization settings
Click this button to configure default visualization settings for images (colormap, interpolation, contrast, etc.).

Results: edit shape styles
Click this button to configure the visual style for annotation shapes displayed on image plots. Parameters are similar to signal shapes but optimized for image visualization (e.g., different colors for better visibility on images).

Results: edit marker styles
Click this button to configure the visual style for cursor markers on image plots.

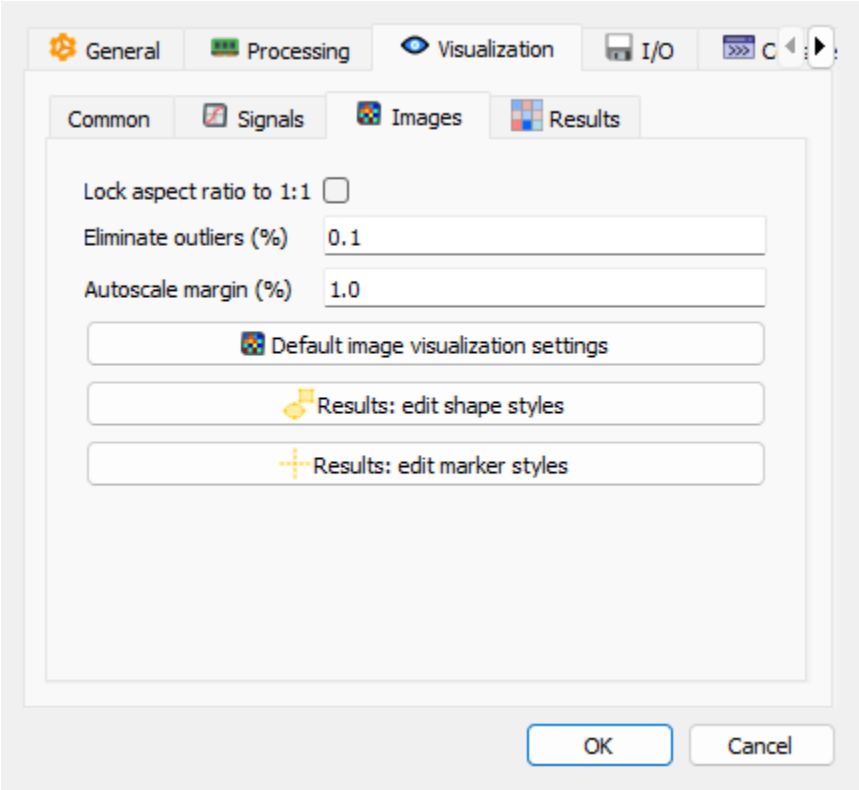


Fig. 7: Visualization settings - Images sub-tab

Results

The Results sub-tab contains settings for displaying analysis results on plots:

These settings control how analysis results are displayed on plots to prevent performance issues with large datasets:

Maximum shapes to draw
: Maximum number of geometry shapes to draw on the plot (default: 1,000). When the number of shapes exceeds this limit, only the first N shapes are drawn and a warning label is displayed.

Maximum cells in label
: Maximum number of table cells (rows × columns) to display in merged result labels on plots (default: 100). When the number of cells exceeds this limit, the table is truncated.

Maximum columns in label
: Maximum number of columns to display in merged result labels (default: 15). When the number of columns exceeds this limit, only the first N columns are displayed.

Show the merged result label by default
: When enabled, the merged result label is shown on the plot by default for new objects. This setting can be toggled per-object using the checkbox in the Properties panel.

Note: These settings affect only the visualization of results on plots. They do not affect the actual computation or storage of results in metadata.

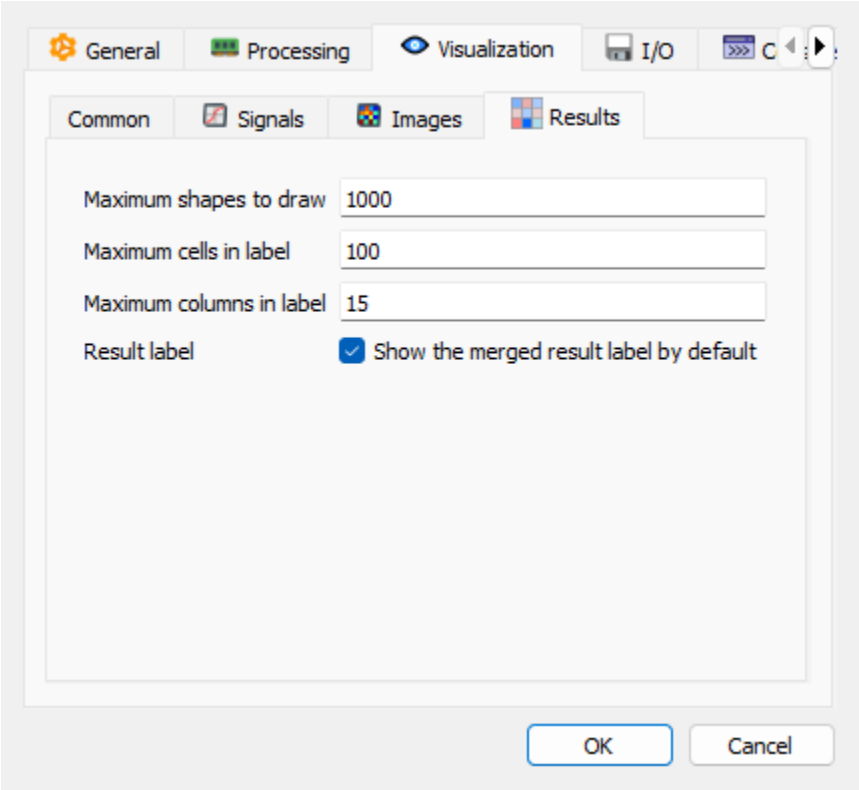


Fig. 8: Visualization settings - Results sub-tab

I/O

The I/O settings tab controls input/output operations:

Clear workspace before loading HDF5 file
When enabled, the workspace is cleared before loading an HDF5 file.

Ask before clearing workspace
When enabled, a confirmation dialog is displayed before clearing the workspace.

HDF5 full path in title
When enabled, the full path of the HDF5 dataset is used as the title for the signal/image object. When disabled, only the dataset name is used.

HDF5 file name in title
When enabled, the HDF5 file name is appended as a suffix to the title of the signal/image object.

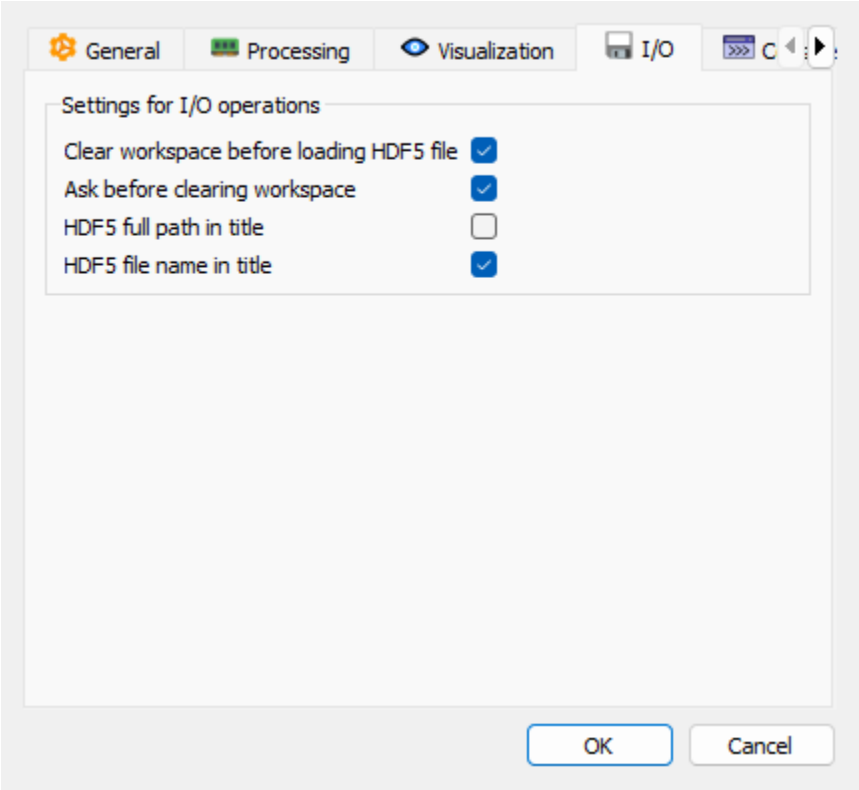


Fig. 9: I/O settings tab

Console

The Console settings tab configures the internal console for debugging and advanced users:

Console enabled
Enable the internal Python console for debugging and advanced scripting.

Show console on error
When enabled, the console is automatically shown when an error occurs in the application. This is useful for debugging as it allows you to see the error traceback.

External editor path
Path to an external text editor to use for editing Python code from the console.

External editor arguments
Command-line arguments to pass to the external editor.

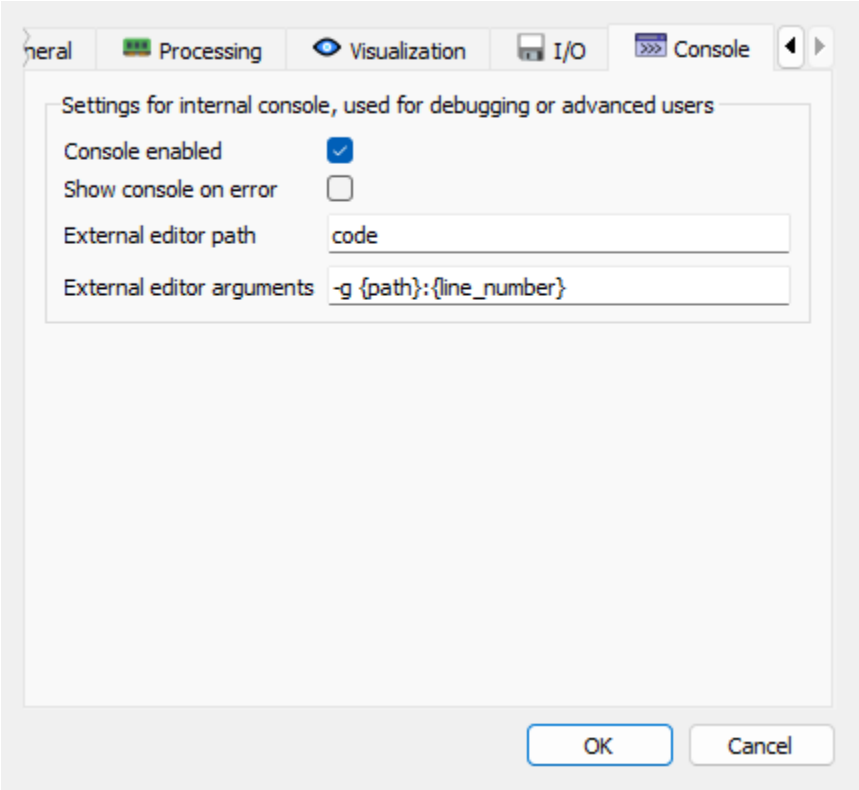
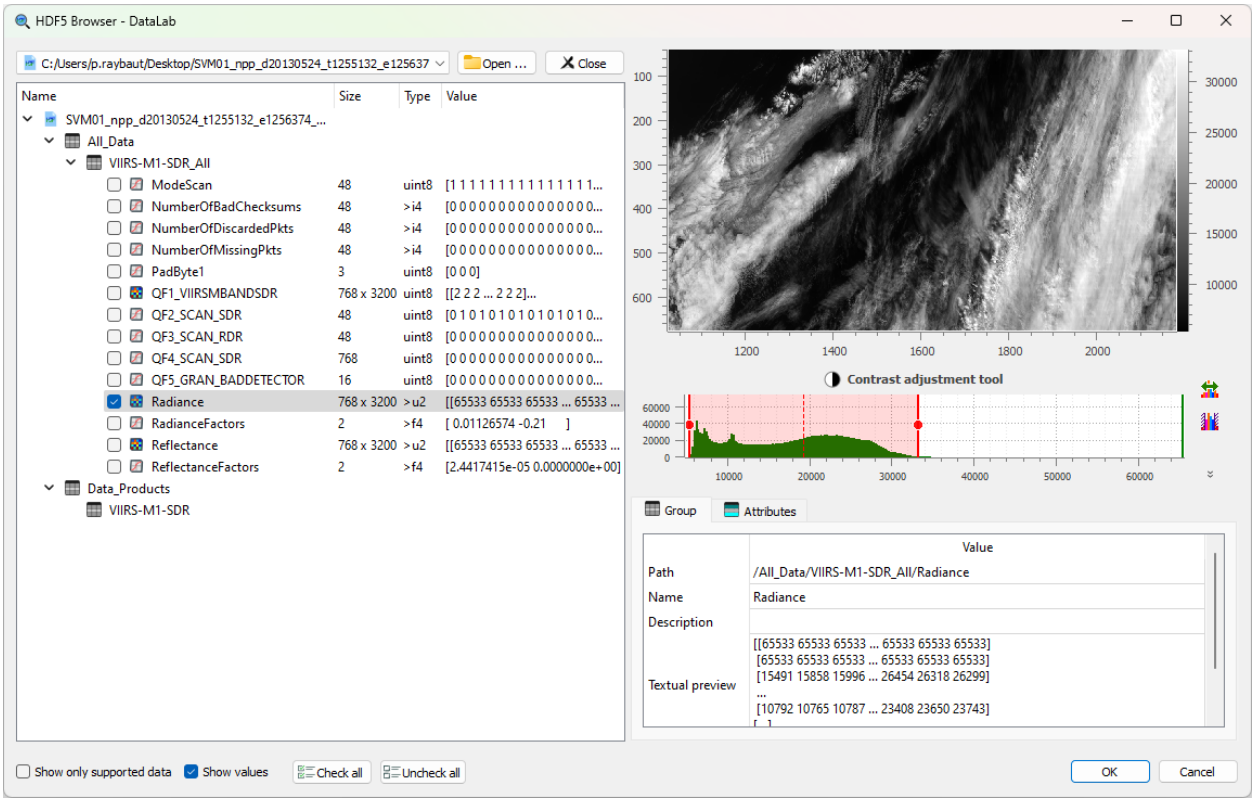


Fig. 10: Console settings tab

2.1.3 HDF5 Browser

The "HDF5 Browser" is a modal dialog box allowing to import almost any 1D and 2D data into DataLab workspace (and eventually metadata).



Compatible curve or image data are displayed in a hierarchical view on the left panel, as well as other scalar data (scalar values are just shown for context purpose and may not be imported into DataLab workspace).

The right panel displays the selected curve or image data. It also shows information on "Group" (path, description, etc.) and "Attributes" (names and values).

The HDF5 browser is fairly simple to use:

- On the left panel, select the curve or image data you want to import
- Selected data is plotted on the right panel
- Click on "Check all" if you want to import all compatible data
- Then validate by clicking on "OK"

Note: The HDF5 browser may be used to explore multiple HDF5 files at once, thus allowing to import data from different files into the same DataLab workspace.

2.2 Signal processing

This section describes the features specific to the signal processing panel. The signal processing panel is the default panel when DataLab is started.

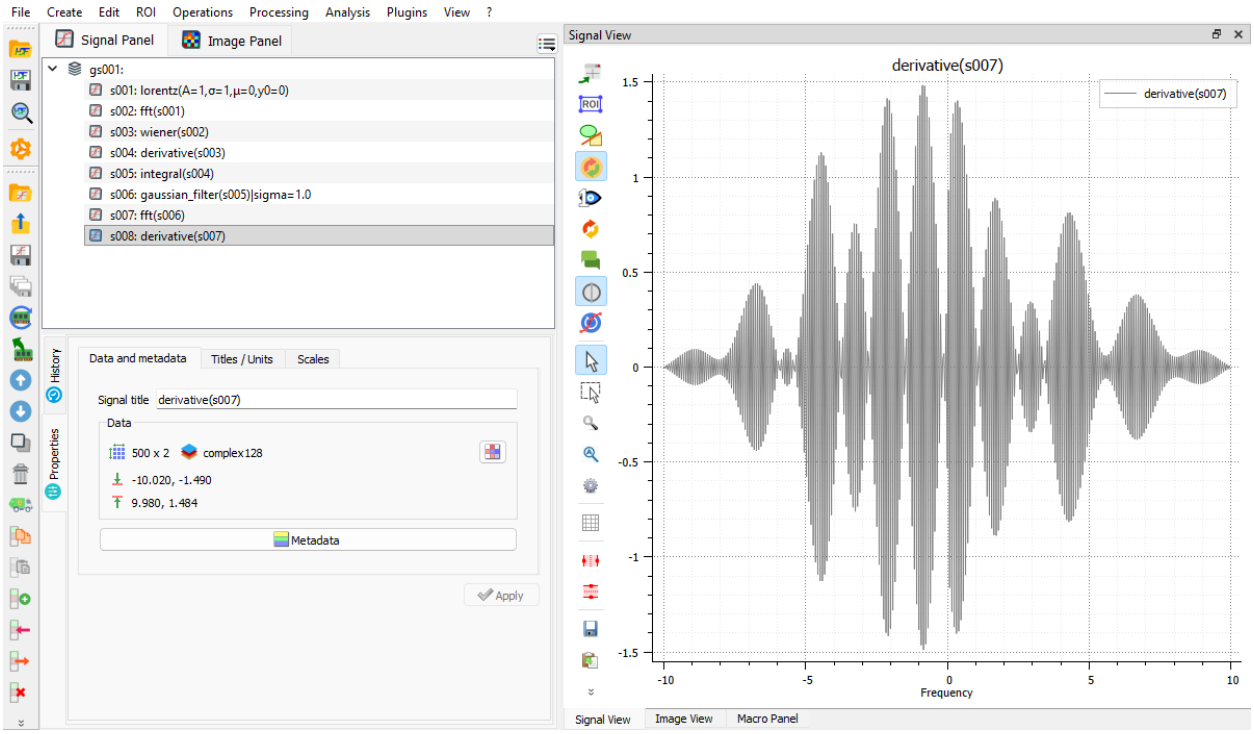


Fig. 11: DataLab main window: Signal processing view

2.2.1 Open and save Signals

This section describes how to open and save signals (and workspaces).

Note: For creating new signals from mathematical models, see *Create Signals*.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "File" menu allows you to:

- Open, save and close signals (see below).
- Save and restore the current workspace or browse HDF5 files (see *Overview*).
- Edit DataLab preferences (see *Settings*).

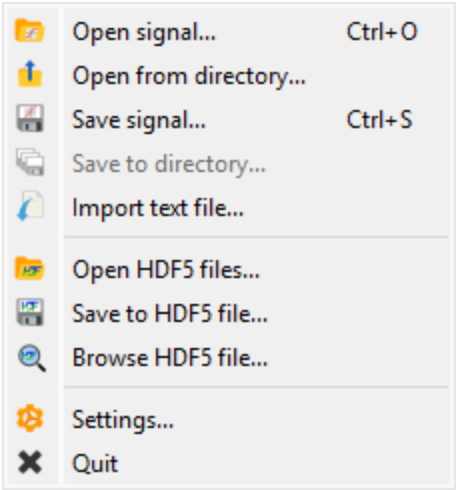


Fig. 12: Screenshot of the "File" menu.

Open signal

Create a new signal from the following supported filetypes:

File type	Extensions
Text files	.txt, .csv
NumPy arrays	.npy
MAT-Files	.mat
FT-Lab files	.sig

Open from directory

Open multiple signals from a specified directory.

Save signal

Save current signal to the following supported filetypes:

File type	Extensions
Text files	.csv

Save signals to directory

Save all selected signals to a specified directory, with configurable filename pattern and signal format.

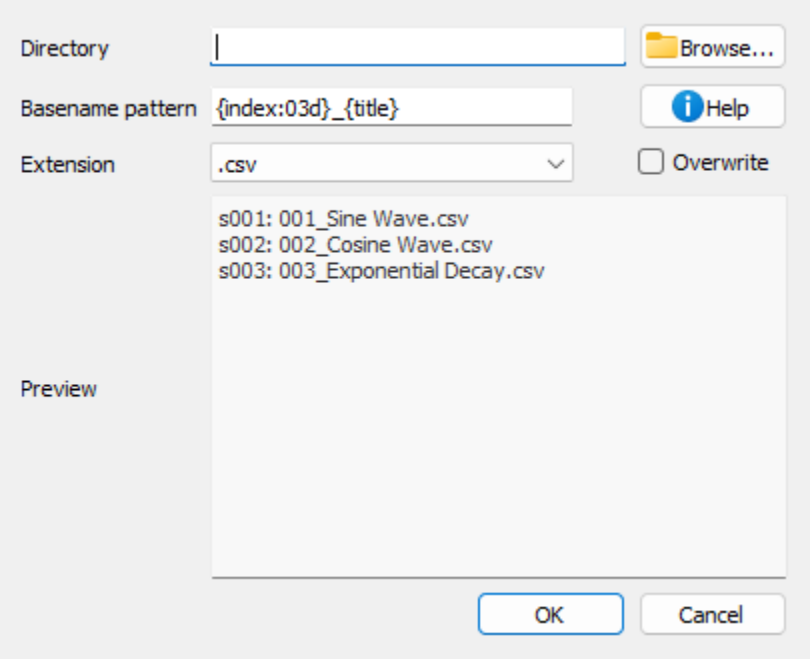


Fig. 13: Save signals to directory dialog.

When you select "Save to directory…" from the File menu, a dialog appears where you can:

- **Directory**: Choose the target directory where signals will be saved
- **Filename pattern**: Define a pattern for the filenames using Python format strings
- **File extension**: Select the output format (.csv, .txt, etc.)
- **Overwrite**: Choose whether to overwrite existing files
- **Preview**: See the list of files that will be created (with object IDs)

The filename pattern supports the following placeholders:

- `{title}`: Signal title
- `{index}`: 1-based index of the signal in the selection (with zero-padding)
- `{count}`: Total number of selected signals
- `{xlabel}`, `{xunit}`, `{ylabel}`, `{yunit}`: Axis labels and units
- `{metadata[key]}`: Access metadata values

You can also use format modifiers, for example `{index:03d}` will format the index with 3 digits zero-padding (001, 002, 003, etc.).

Import text file

DataLab can natively import signal files (e.g. CSV, NPY, etc.). However some specific text file formats may not be supported. In this case, you can use the *Import text file* feature, which allows you to import a text file and convert it to a signal.

This feature is accessible from the *File* menu, under the *Import text file* option.

It opens an import wizard that guides you through the process of importing the text file.

Step 1: Select the source

The first step is to select the source of the text file. You can either select a file from your computer or the clipboard if you have copied the text from another application.

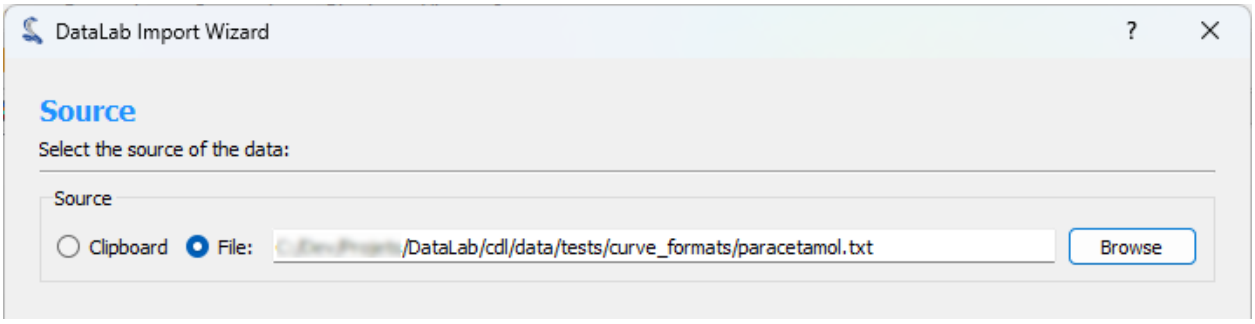


Fig. 14: Step 1: Select the source

Step 2: Preview and configure the import

The second step consists of configuring the import and previewing the result. You can configure the following options:

- **Delimiter**: The character used to separate the values in the text file.
- **Comments**: The character used to indicate that the line is a comment and should be ignored.
- **Rows to Skip**: The number of rows to skip at the beginning of the file.
- **Maximum Number of Rows**: The maximum number of rows to import. If the file contains more rows, they will be ignored.
- **Transpose**: If checked, the rows and columns will be transposed.
- **Data type**: The destination data type of the imported data.
- **First Column is X**: If checked, the first column will be used as the X axis.

When you are done configuring the import, click the *Apply* button to see the result.

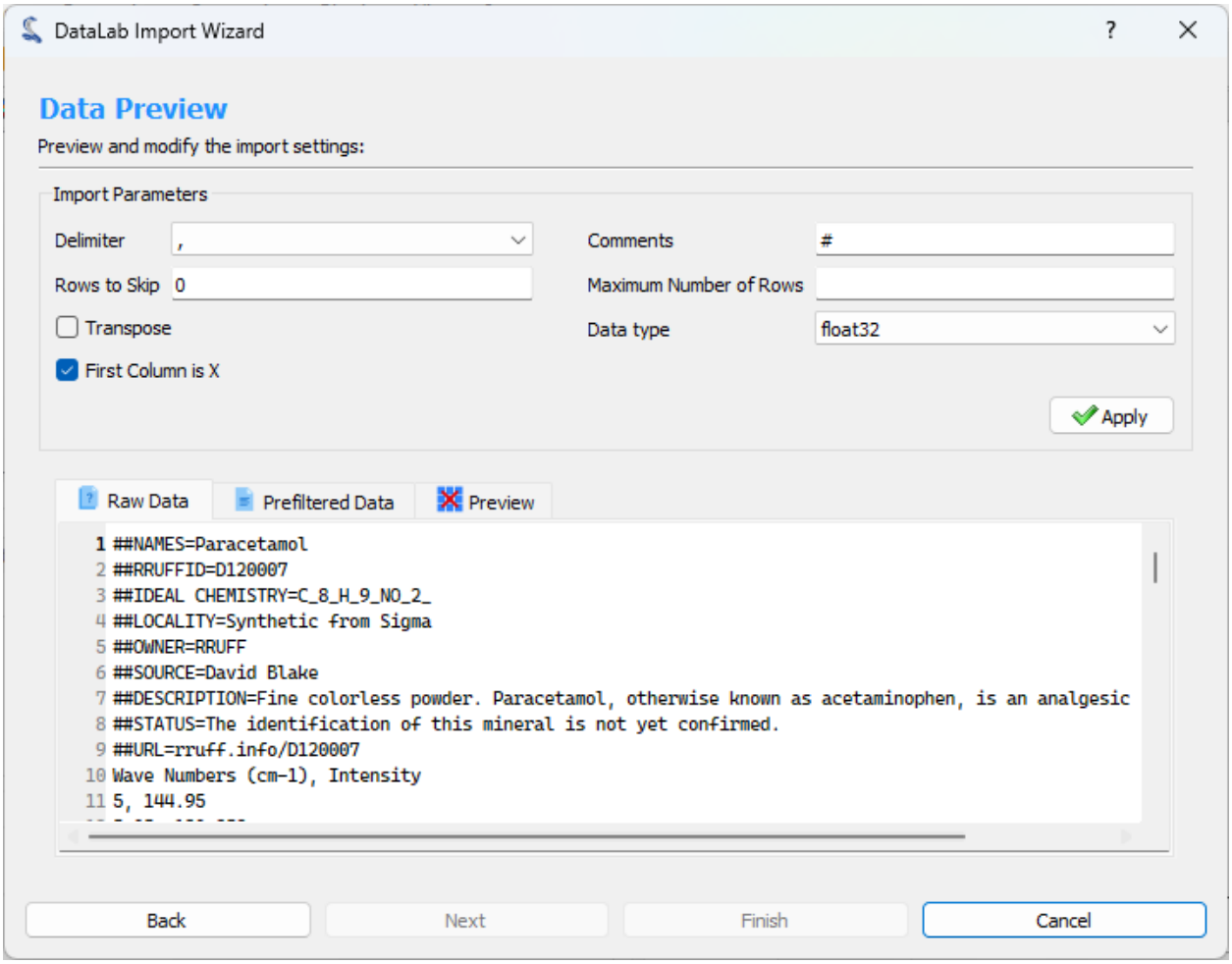


Fig. 15: Step 2: Configure the import

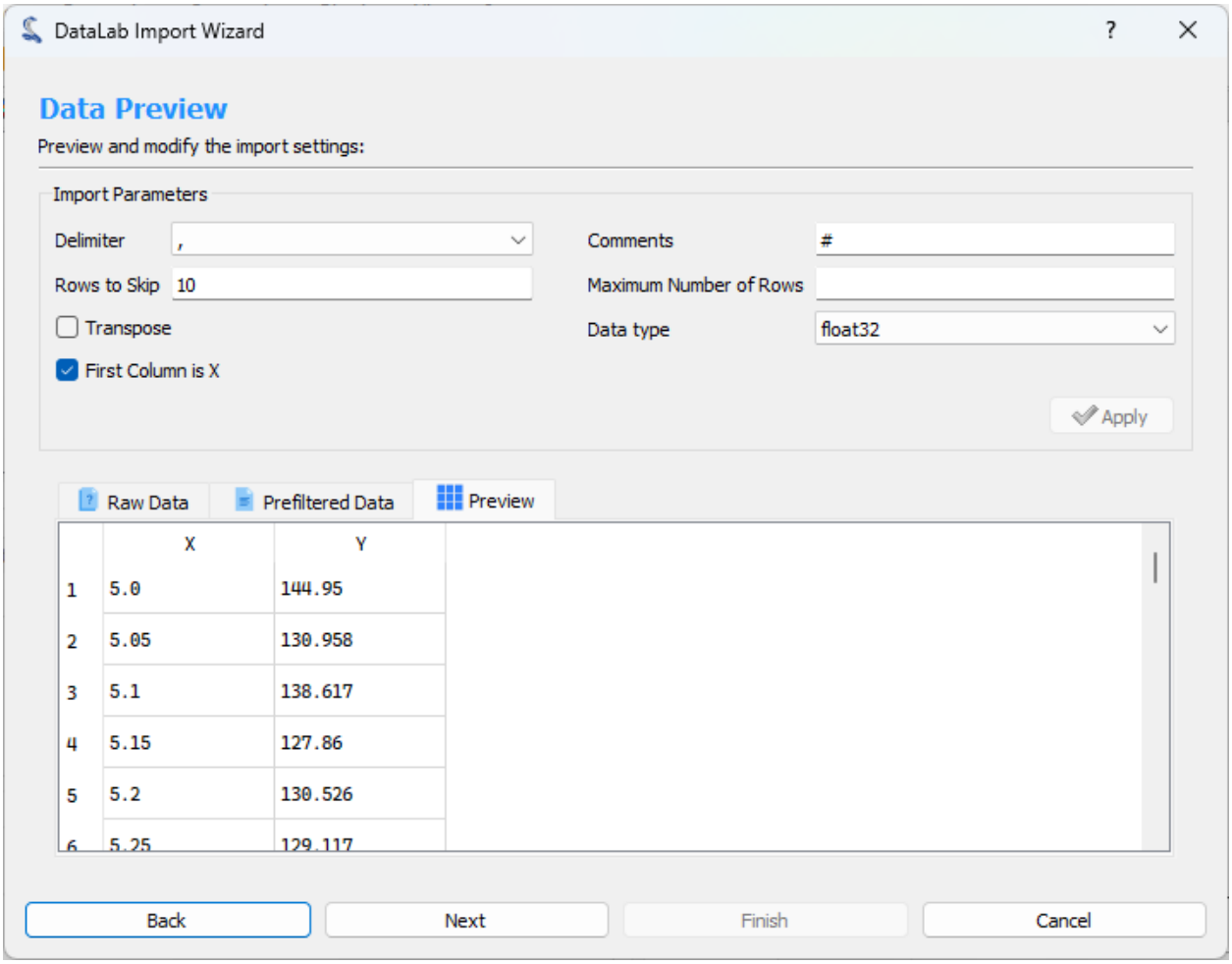


Fig. 16: Step 2: Preview the result

Step 3: Show graphical representation

The third step shows a graphical representation of the imported data. You can use the *Finish* button to import the data into DataLab workspace.

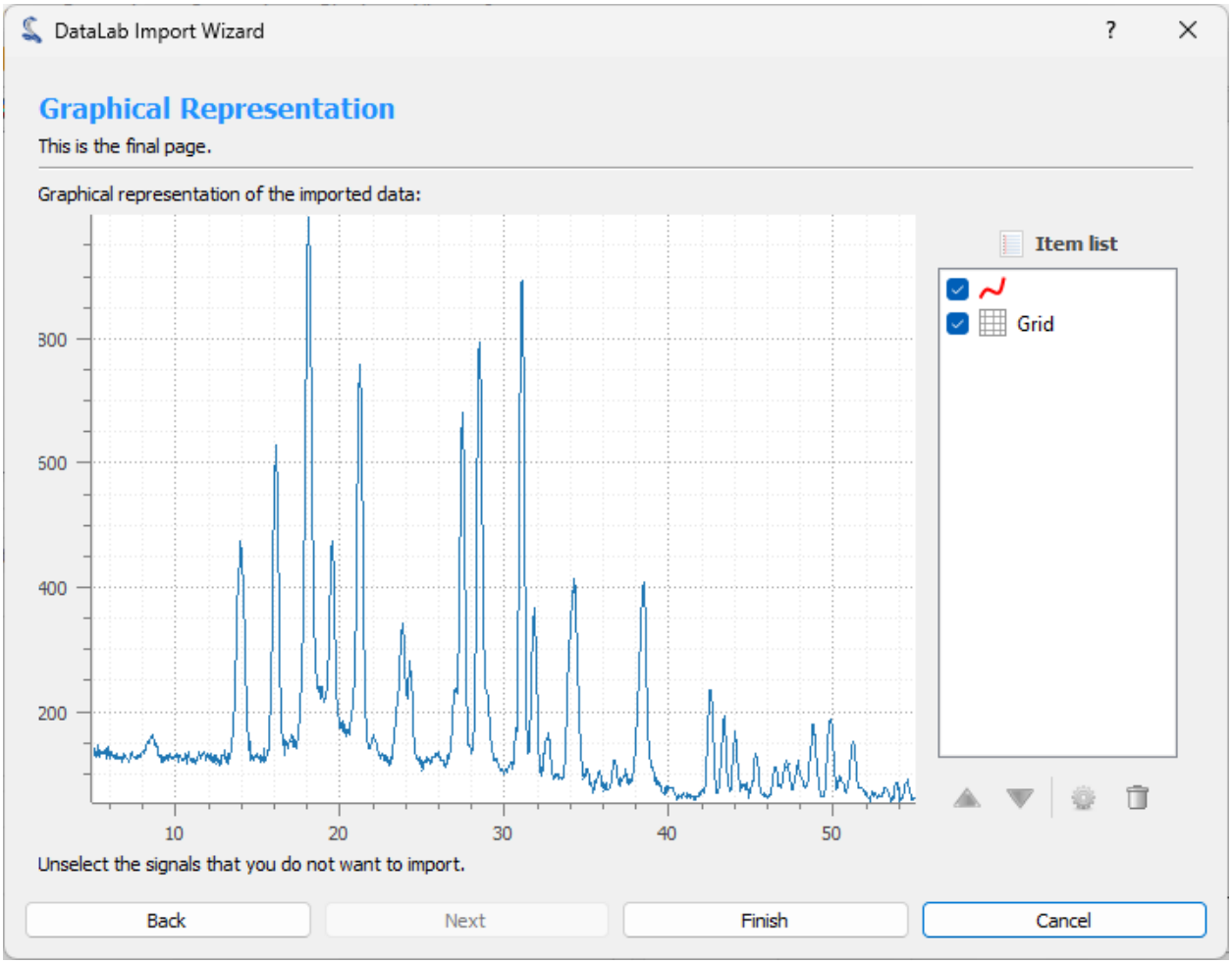


Fig. 17: Step 3: Show graphical representation

2.2.2 Create Signals

This section describes how to create new signals from various mathematical models.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "Create" menu allows you to create new signals from various models (see below).

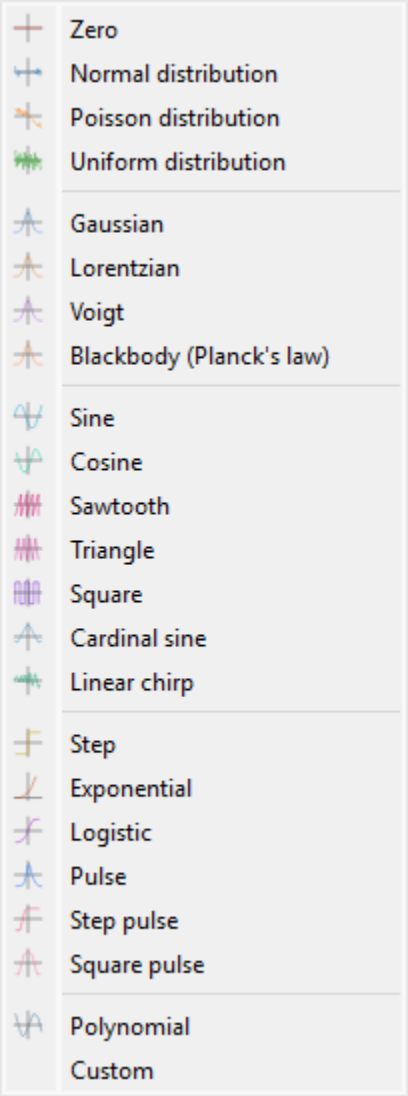


Fig. 18: Screenshot of the "Create" menu.

New signal

Create a new signal from various models:

Icon	Model	Equation
	Zero	$y[i] = 0$
	Normal distribution	$y[i]$ is normally distributed with configurable mean and standard deviation
	Poisson distribution	$y[i]$ is Poisson distributed with configurable mean
	Uniform distribution	$y[i]$ is uniformly distributed between two configurable bounds
	Gaussian	$y = y_0 + \frac{A}{\sqrt{2\pi} \cdot \sigma} \cdot \exp\left(-\frac{1}{2} \cdot \left(\frac{x - x_0}{\sigma}\right)^2\right)$
	Lorentzian	$y = y_0 + \frac{A}{\sigma \cdot \pi} \cdot \frac{1}{1 + \left(\frac{x - x_0}{\sigma}\right)^2}$
	Voigt	$y = y_0 + A \cdot \frac{\Re\left(\exp\left(-z^2\right) \cdot \mathrm{erfc}(-j \cdot z)\right)}{\sqrt{2\pi} \cdot \sigma}$ with $z = \frac{x - x_0 - j \cdot \sigma}{\sqrt{2} \cdot \sigma}$
	Blackbody (Planck's law)	$y = \frac{2hc^2}{\lambda^5 \left(\exp\left(\frac{hc}{\lambda kT}\right) - 1\right)}$
	Sine	$y = y_0 + A \sin\left(2\pi \cdot f \cdot x + \phi\right)$
	Cosine	$y = y_0 + A \cos\left(2\pi \cdot f \cdot x + \phi\right)$
	Sawtooth	$y = y_0 + A\left(2\left(fx + \frac{\phi}{2\pi} - \left\lfloor fx + \frac{\phi}{2\pi} + \frac{1}{2}\right\rfloor\right)\right)$
	Triangle	$y = y_0 + A\,\mathrm{sawtooth}\left(2\pi f x + \phi, \mathrm{width} = 0.5\right)$
	Square	$y = y_0 + A\,\mathrm{sgn}\left(\sin\left(2\pi f x + \phi\right)\right)$
	Cardinal sine	$y = y_0 + A\,\mathrm{sinc}\left(2\pi f x + \phi\right)$
	Linear chirp	$y = y_0 + A \sin\left(\phi_0 + 2\pi\left(f_0\, x + \frac{1}{2} c\, x^2\right)\right)$
	Step	$y = y_0 + A \begin{cases} 1 & \text{if } x > x_0 \\ 0 & \text{otherwise} \end{cases}$
	Exponential	$y = y_0 + A \exp\left(B \cdot x\right)$
	Logistic	$y = y_0 + \frac{A}{1 + \exp\left(-k\left(x - x_0\right)\right)}$
	Pulse	$y = y_0 + A \begin{cases} 1 & \text{if } x_0 < x < x_1 \\ 0 & \text{otherwise} \end{cases}$
	Step Pulse	$y = \left(\begin{cases} y_0 & \text{if } x < t_0 \\ y_0 + A \cdot \frac{x - t_0}{t_r} & \text{if } t_0 \le x < t_0 + t_r \\ y_0 + A & \text{if } x \ge t_0 + t_r \end{cases}\right) + \mathcal{N}\left(0, \sigma_n\right)$ where: • t_0 is the pulse start time, • t_r is the rise time, • σ_n is the noise amplitude

continues on next page

Table 1 – continued from previous page

Icon	Model	Equation
	Square Pulse	$y(x) = \left(\begin{cases} y_0 & \text{if } x < t_0 \\ y_0 + A \cdot \dfrac{x - t_0}{t_r} & \text{if } t_0 \leq x < t_0 + t_r \\ y_0 + A & \text{if } t_0 + t_r \leq x < t_1 \\ y_0 + A - A \cdot \dfrac{x - t_1}{t_f} & \text{if } t_1 \leq x < t_1 + t_f \\ y_0 & \text{if } x \geq t_1 + t_f \end{cases} \right) + \mathcal{N}(0, \sigma_n)$ where: • t_0 is the pulse start time, • t_r is the rise time, • t_f is the fall time, • $t_1 = t_0 + t_r + d$ is the time at which the decay starts, • σ_n is the noise amplitude • the duration of the plateau d is computed as $d = t_{\mathrm{FWHM}} - \dfrac{t_r + t_f}{2}$ from the full width at half maximum t_{FWHM} **Warning:** The duration of the plateau d should not be negative.
	Polynomial	$y = y_0 + A_0 + A_1 \cdot x + A_2 \cdot x^2 + \ldots + A_n \cdot x^n$
	Custom	Manual input of X and Y values

2.2.3 Manipulate metadata and annotations

This section describes how to manipulate metadata and annotations in DataLab.

Signal metadata contains various information about the signal or its representation, such as view settings, Regions Of Interest (ROIs), processing chain history, analysis results, and any other information that you may have added to the metadata of a signal (or that comes from the signal file itself).

The "Edit" menu allows you to perform classic editing operations on the current signal or group of signals (create/rename group, move up/down, delete signal/group of signals, etc.).

As detailed below, it also allows you to:

- Navigate and utilize the processing chain history through actions like "Recompute" and "Select source objects".
- Manipulate metadata and annotations associated with the current signal, thanks to the "Metadata" and "Annotations" submenus which provide the following features.

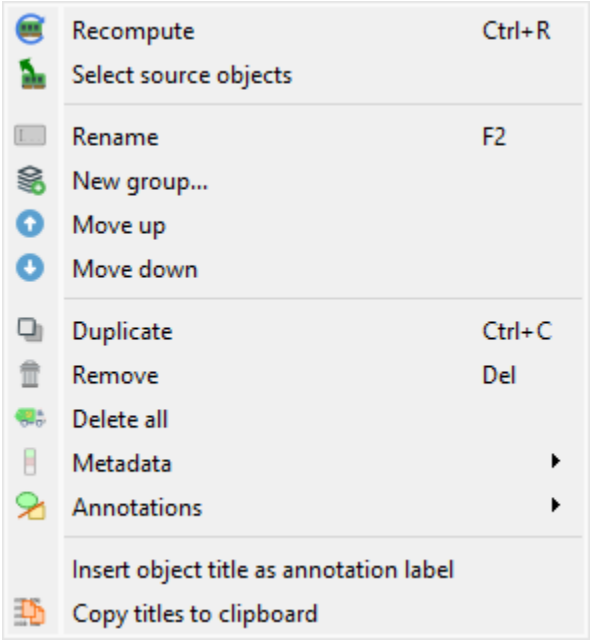


Fig. 19: Screenshot of the "Edit" menu.

Recompute

The "Recompute" action allows you to recompute the selected signal(s) using their original processing parameters. This is useful when you want to re-execute the processing chain that was used to create a signal, for example after modifying global settings or dependencies.

Note: This action is only available for signals that were created through processing operations and have stored processing parameters.

Select source objects

The "Select source objects" action allows you to select the source object(s) that were used to create the currently selected signal. This helps trace back the processing history and understand which original signals were used as input for the current result.

Note: This action is only available when exactly one signal is selected and that signal has source object references.

Metadata

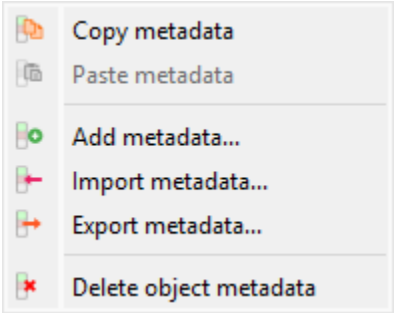


Fig. 20: Screenshot of the "Metadata" submenu.

Copy/paste metadata

As metadata contains useful information about the signal, it can be copied and pasted from one signal to another by selecting the "Copy metadata" and "Paste metadata" actions in the "Edit" menu.

This feature allows you to tranfer those information from one signal to another:

- *Regions Of Interest (ROIs)*: that is a very efficient way to reuse the same ROI on different signals and easily compare the results of the analysis on those signals
- Analyze results, such as peak positions or FHWM intervals (the relevance of transferring such information depends on the context and is up to the user to decide)
- Any other information that you may have added to the metadata of a signal

Note: Copying metadata from a signal to another will overwrite the metadata of the destination signal (for the metadata keys that are common to both signals) or simply add the metadata keys that are not present in the destination signal.

Import/export metadata

Metadata can also be imported and exported from/to a JSON file using the "Import metadata" and "Export metadata" actions in the "Edit" menu. This is exactly the same as the copy/paste metadata feature (see above for more details on the use cases of this feature), but it allows you to save the metadata to a file and then import it back later.

Delete metadata

When deleting metadata using the "Delete metadata" action in the "Edit" menu, you will be prompted to confirm the deletion of Region of Interests (ROIs) if they are present in the metadata. After this eventual confirmation, the metadata will be deleted, meaning that analysis results, ROIs, and any other information associated with the signal will be lost.

Add metadata

The "Add metadata" action allows you to add custom metadata items to one or more selected signals. This is useful for tagging signals with experiment IDs, sample names, processing steps, or any other custom information.

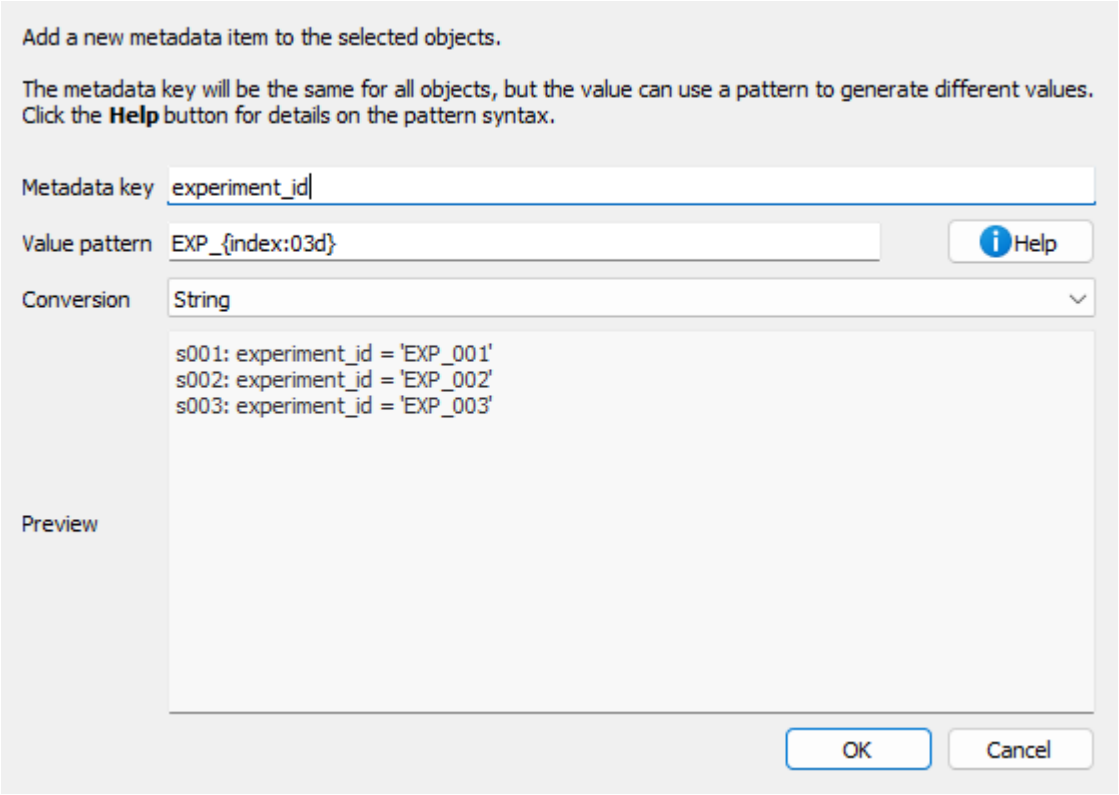


Fig. 21: Add metadata dialog.

When you select "Add metadata…" from the Edit menu, a dialog appears where you can:

- **Metadata key**: Enter the name of the metadata field to add
- **Value pattern**: Define a pattern for the metadata value using Python format strings
- **Conversion**: Choose how to store the value (string, float, integer, or boolean)
- **Preview**: See how the metadata will be added to each selected signal

The value pattern supports the following placeholders:

- `{title}`: Signal title
- `{index}`: 1-based index of the signal in the selection
- `{count}`: Total number of selected signals
- `{xlabel}`, `{xunit}`, `{ylabel}`, `{yunit}`: Axis labels and units
- `{metadata[key]}`: Access existing metadata values

You can also use format modifiers:

- `{title:upper}`: Convert to uppercase
- `{title:lower}`: Convert to lowercase

- `{index:03d}`: Format as 3-digit number with leading zeros

Examples:

- Add experiment ID: key=``experiment_id``, pattern=``EXP_{index:03d}``, conversion=string → Creates metadata like `experiment_id="EXP_001"`
- Add sample temperature: key=``temperature``, pattern=``{metadata[temp]}``, conversion=float → Copies temperature from existing metadata and converts to float
- Mark processed signals: key=``is_processed``, pattern=``true``, conversion=bool → Sets `is_processed=True` for all selected signals

Annotations

Annotations are visual elements that can be added to signals to highlight specific features, mark regions of interest, or add explanatory notes. DataLab provides a dedicated submenu in the "Edit" menu for managing annotations.

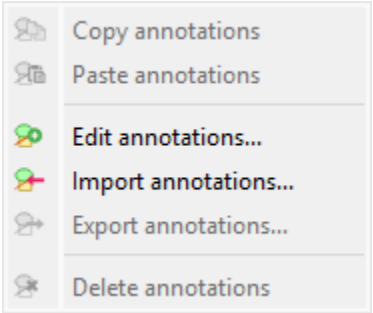


Fig. 22: Screenshot of the "Annotations" submenu.

Copy/paste annotations

Annotations can be copied from one signal and pasted to one or more other signals using the "Copy annotations" and "Paste annotations" actions. This is useful when you want to apply the same visual markers across multiple signals.

The "Paste annotations" action is only enabled when there are annotations in the clipboard (i.e., after using "Copy annotations").

Edit annotations

The "Edit annotations" action opens a dialog where you can view, add, modify, or remove annotations from the current signal. This provides a visual way to manage all annotations on a signal.

Import/export annotations

Annotations can be saved to and loaded from JSON files (.dlabann extension) using the "Import annotations" and "Export annotations" actions. This allows you to:

- Save annotation sets for later reuse
- Share annotations with colleagues
- Archive annotations separately from signal data
- Apply the same annotations to different signals across sessions

The "Export annotations" action is only available when the selected signal has annotations.

Delete annotations

The "Delete annotations" action removes all annotations from the selected signal(s). This action is only enabled when the selected signal(s) have annotations.

Note: Annotations are stored separately from metadata and analysis results. Deleting annotations does not affect ROIs or other metadata items.

Signal titles

Signal titles may be considered as metadata from a user point of view, even if they are not stored in the metadata of the signal (but in an attribute of the signal object).

The "Edit" menu allows you to:

- "Add object title to plot": this action will add a label on top of the signal with its title.
- "Copy titles to clipboard" : this action will copy the titles of the selected signals to the clipboard, which might be useful to paste them in a text editor or in a spreadsheet.

  Example of the content of the clipboard:

```
g001:
    s001: lorentz(a=1,sigma=1,mu=0,ymin=0)
    s002: derivative(s001)
    s003: wiener(s002)
g002: derivative(g001)
    s004: derivative(s001)
    s005: derivative(s002)
    s006: derivative(s003)
g003: fft(g002)
    s007: fft(s004)
    s008: fft(s005)
    s009: fft(s006)
```

2.2.4 Regions Of Interest (ROI)

This section describes how to manipulate Regions Of Interest (ROIs) for signals in DataLab.

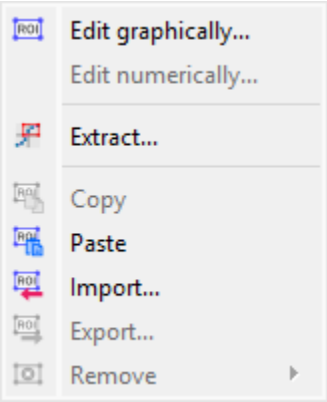


Fig. 23: Screenshot of the "ROI" menu.

The "ROI" menu allows you to manage Regions Of Interest (ROIs) associated with the current signal.

See also:

For more information about signal metadata, see the *Manipulate metadata and annotations* section.

The Regions Of Interest (ROI) are signal areas that are defined by the user to perform specific operations, processing, or analysis on them.

ROI are taken into account almost in all computing features in DataLab:

- The "Operations" menu features are done only on the ROI if one is defined (except if the operation changes the number of points, like interpolation or resampling).
- The "Processing" menu actions are performed only on the ROI if one is defined (except if the destination signal data type is different from the source's, like in the Fourier analysis features).
- The "Analysis" menu actions are done only on the ROI if one is defined.

Note: ROI are stored as metadata, and thus attached to signal.

The "ROI" menu allows you to:

- "Edit graphically" : open a dialog box to manage ROI associated with the selected signal (add, remove, move, resize, etc.). The ROI definition dialog is exactly the same as ROI extraction (see below): the ROI is defined by moving the position and adjusting the width of an horizontal range.
- "Edit numerically": open a dialog box to edit the parameters of the selected ROIs numerically (i.e. using a simple form). This allows you to define or modify ROIs based on numerical values.
- "Extract" : extract the defined ROI from the selected signals. This will create a new signal for each ROI (or a single signal, if the "Extract all ROIs into a single signal" option is selected in the dialog), with the same metadata as the original signal, but with the data corresponding to the ROI only. The new signals will be added to the current workspace.
- "Copy" : copy the defined ROI from the selected signals to the clipboard. This allows you to paste the ROI into another signal or use it in other operations.
- "Paste" : paste the copied ROI from the clipboard to the selected signals. This will add the ROI to the signals, allowing you to define or modify ROIs based on previously copied ones.

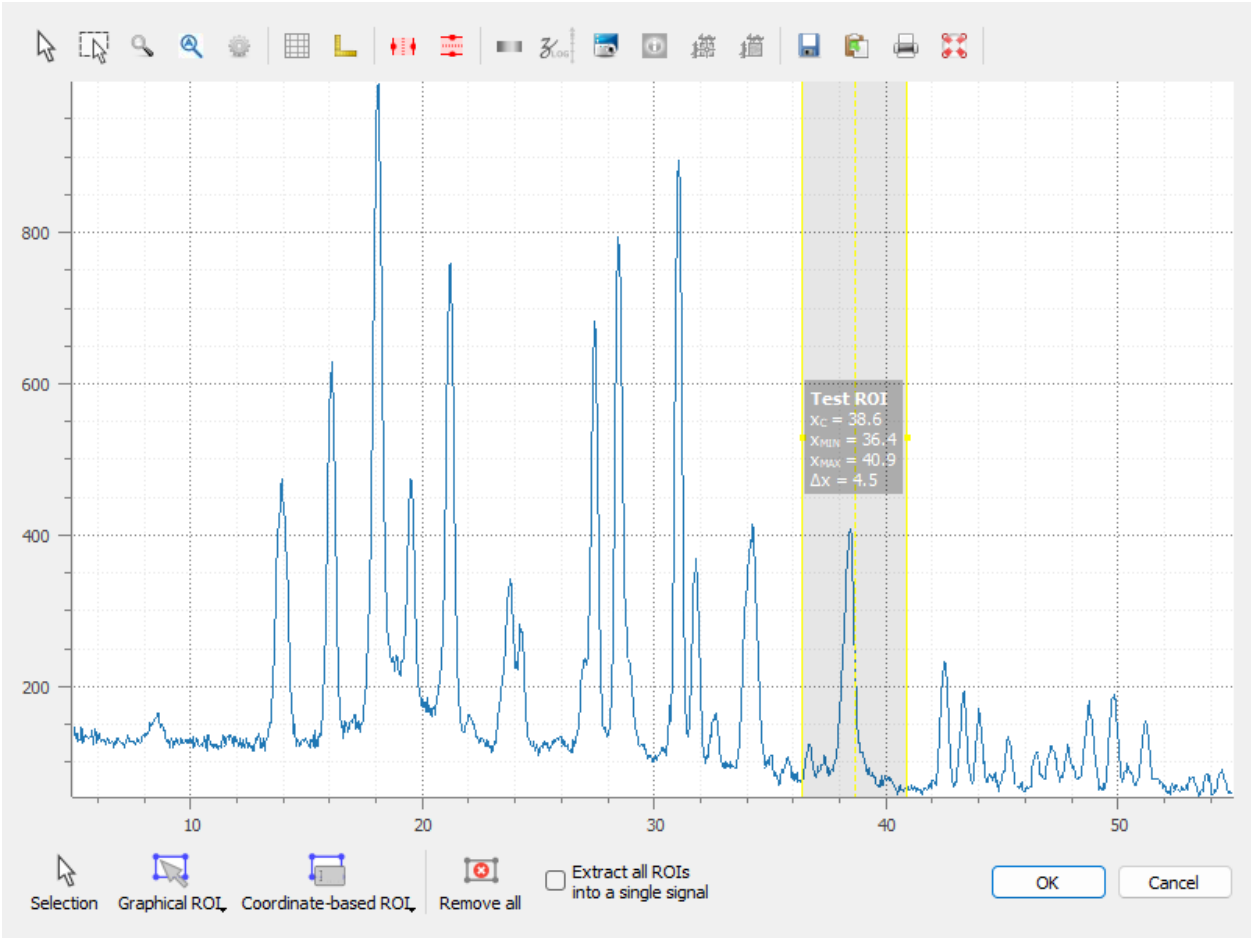


Fig. 24: A signal with an ROI.

- "Import" : import ROIs from a file. This allows you to load previously saved ROIs into the current signal.
- "Export" : export the defined ROIs to a file. This allows you to save the ROIs for later use or to share them with others.
- "Remove all" : remove all defined ROI for the selected signals.

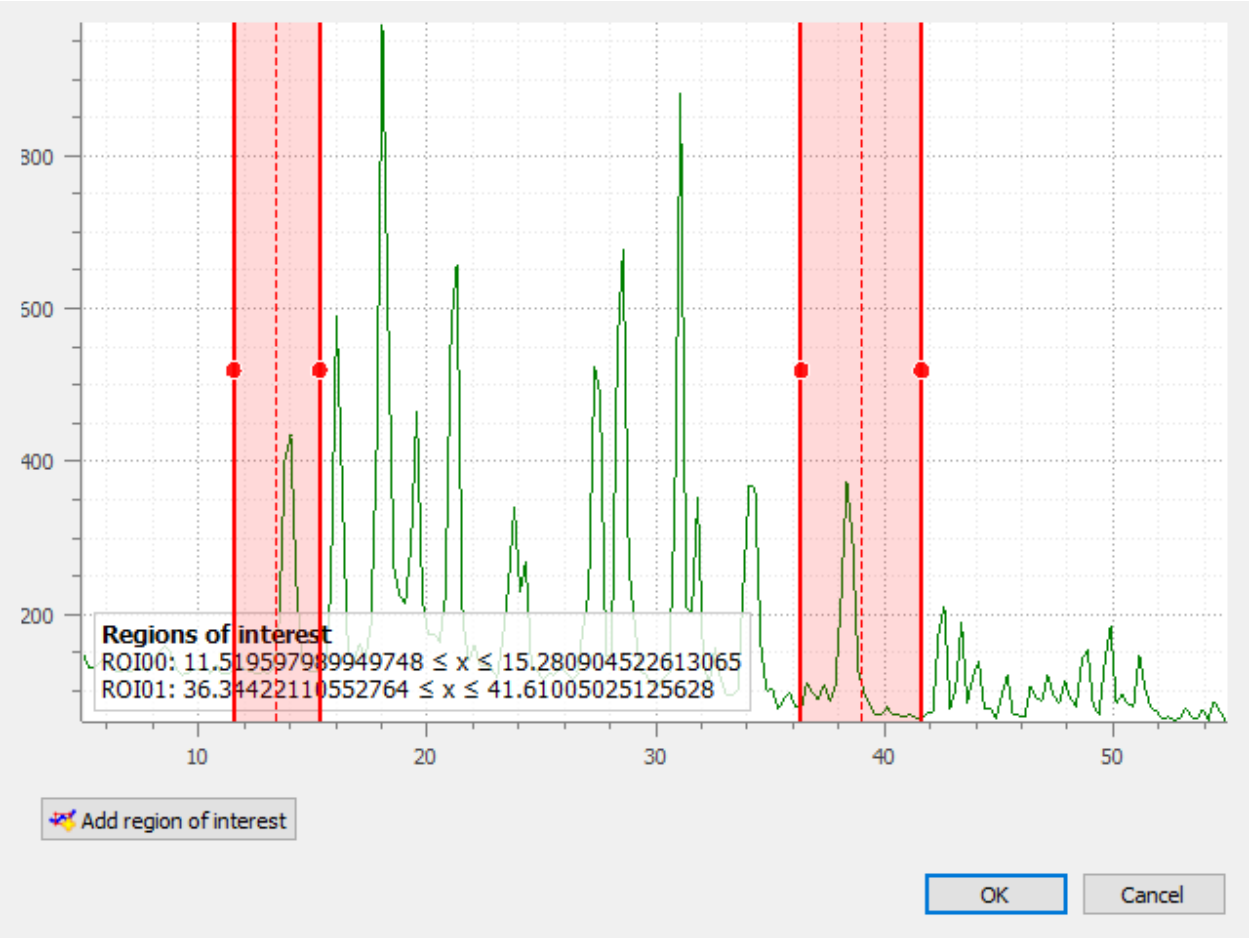


Fig. 25: ROI extraction dialog: the ROI is defined by moving the position and adjusting the width of an horizontal range.

2.2.5 Operations on Signals

This section describes the operations that can be performed on signals.

See also:

Processing Signals for more information on signal processing features, or *Analysis features on Signals* for information on analysis features on signals.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "Operations" menu allows you to perform various operations on the selected signals, such as arithmetic operations, peak detection, or convolution.

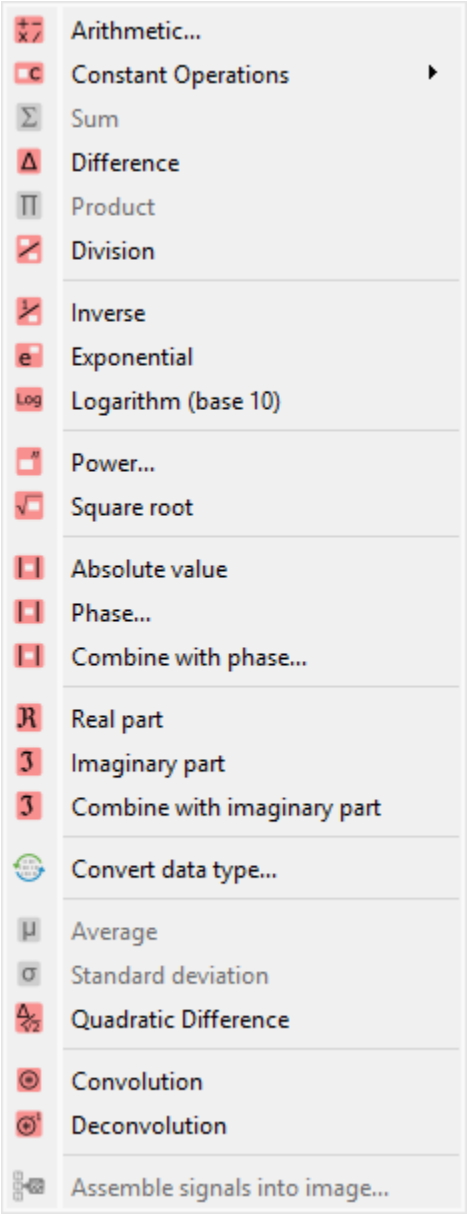


Fig. 26: Screenshot of the "Operations" menu.

Operations with a constant

Create a new signal which is the result of a constant operation on each selected signal:

Operation	Description
Addition	$y_k = y_{k-1} + c$
Subtraction	$y_k = y_{k-1} - c$
Multiplication	$y_k = y_{k-1} \times c$
Division	$y_k = \frac{y_{k-1}}{c}$

Basic arithmetic operations

Operation	Description
Sum	$y_M = \sum_{k=0}^{M-1} y_k$
Difference	$y_2 = y_1 - y_0$
Product	$y_M = \prod_{k=0}^{M-1} y_k$
Division	$y_2 = \frac{y_1}{y_0}$
Inverse	$y_2 = \frac{1}{y_1}$
Exponential	$y_2 = \exp(y_1)$
Logarithm (base 10)	$y_2 = \log_{10}(y_1)$

Convolution and Deconvolution

Operation	Implementation
Convolution	Based on scipy.signal.convolve
Deconvolution	Frequency domain deconvolution

Absolute value and complex signal operations

Operation	Description
Absolute value	$y_k = \lvert y_{k-1} \rvert$
Phase (argument)	`sigima.proc.signal.phase()`
Combine with phase	Consider current signal as the module and allow to select a signal representing the phase to merge them in a complex signal `sigima.proc.signal.complex_from_magnitude_phase()`
Real part	$y_k = \Re(y_{k-1})$
Imaginary part	$y_k = \Im(y_{k-1})$
Combine with imaginary part	Consider current signal as the real part and allow to select a signal representing the imaginary part to merge them in a complex signal `sigima.proc.signal.complex_from_real_imag()`

Data type conversion

The "Convert data type" action allows you to convert the data type of the selected signals.

Note: Data type conversion relies on `numpy.ndarray.astype()` function with the default parameters (*casting='unsafe'*).

Statistics between signals

Create a new signal which is the result of a statistical operation on each point of the selected signals:

Operation	Description
Average	$y_M = \frac{1}{M}\sum_{k=0}^{M-1} y_k$
Standard Deviation	$y_M = \sqrt{\frac{1}{M}\sum_{k=0}^{M-1} (y_k - \bar{y})^2}$

Other mathematical functions

Function	Description
Power	$y_k = y_k^n$
Square root	$y_k = \sqrt{y_k}$
Derivative	Based on numpy.gradient
Integral	Based on scipy.integrate.cumulative_trapezoid
Assemble signals into image	Create a 2D image by assembling selected 1D signals as rows or columns, with optional normalization.

2.2.6 Processing Signals

This section describes the signal processing features available in DataLab.

See also:

Operations on Signals for more information on operations that can be performed on signals, or *Analysis features on Signals* for information on analysis features on signals.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "Processing" menu allows you to perform various processing on the selected signals, such as smoothing, normalization, or interpolation.

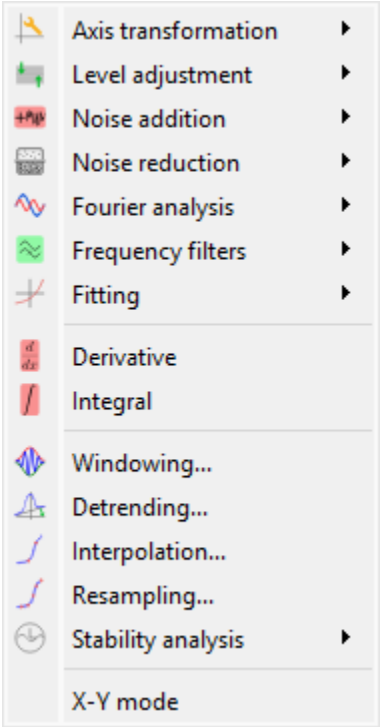


Fig. 27: Screenshot of the "Processing" menu.

Axis transformation

Linear calibration

Create a new signal which is a linear calibration of each selected signal with respect to X or Y axis:

Parameter	Linear calibration
X-axis	$x_1 = a.x_0 + b$
Y-axis	$y_1 = a.y_0 + b$

Swap X/Y axes

Create a new signal which is the result of swapping X/Y data.

Reverse X-axis

Create a new signal which is the result of reversing X data.

Convert to Cartesian coordinates

Create a new signal which is the result of converting polar coordinates to Cartesian coordinates.

This function assumes that the x-axis represents the radius and the y-axis the angle.

Warning: A radius cannot be negative. Any negative value is clipped to 0.

Convert to polar coordinates

Create a new signal which is the result of converting Cartesian coordinates to polar coordinates.

Level adjustment

Normalize

Create a new signal which is the normalization of each selected signal by maximum, amplitude, sum, energy or RMS:

Parameter	Normalization
Maximum	$y_1 = \frac{y_0}{\max(y_0)}$
Amplitude	$y_1 = \frac{y'_0}{\max(y'_0)}$ with $y'_0 = y_0 - \min(y_0)$
Area	$y_1 = \frac{y_0}{\sum_{n=0}^{N} y_0[n]}$
Energy	$y_1 = \frac{y_0}{\sqrt{\sum_{n=0}^{N} \lvert y_0[n]\rvert^2}}$
RMS	$y_1 = \frac{y_0}{\sqrt{\frac{1}{N}\sum_{n=0}^{N} \lvert y_0[n]\rvert^2}}$

Clipping

Create a new signal which is the result of clipping each selected signal.

Offset correction

Create a new signal which is the result of offset correction of each selected signal. This operation is performed by subtracting the signal baseline which is estimated by the mean value of a user-defined range.

Noise addition

Generate new signals by adding the same noise to each selected signal. The available noise types are:

Noise	Description
Gaussian	Normal distribution
Uniform	Uniform distribution
Poisson	Poisson distribution

Noise reduction

Create a new signal which is the result of noise reduction of each selected signal.

The following filters are available:

Filter	Formula/implementation
Gaussian filter	scipy.ndimage.gaussian_filter
Moving average	scipy.ndimage.uniform_filter
Moving median	scipy.ndimage.median_filter
Wiener filter	scipy.signal.wiener

Fourier analysis

Zero padding

Create a new signal which is the result of zero padding of each selected signal.

The following parameters are available:

Parameter	Description
Strategy	Zero padding strategy (see below)
Number of points	Custom length (if *strategy* is "custom")

Zero padding strategy refers to the method used to increase the length of the signal, and it can be one of the following:

Strategy	Description
next_pow2	Next power of 2 (e.g. 1024 → 2048)
double	Double the length (e.g. 1024 → 2048)
triple	Triple the length (e.g. 1024 → 3072)
custom	Custom length (user-defined)

FFT related functions

Create a new signal which is the result of a Fourier analysis of each selected signal.

The following functions are available:

Function	Description	Formula/implementation
FFT	Fast Fourier Transform	numpy.fft.fft
Inverse FFT	Inverse Fast Fourier Transform	numpy.fft.ifft
Magnitude spectrum	Optional: output in decibels (dB)	$y_1 = \lvert\mathrm{FFT}\left(y_0\right)\rvert$ or $y_1 = 20\log_{10}\left(\lvert\mathrm{FFT}\left(y_0\right)\rvert\right)$ (dB)
Phase spectrum	Phase of the FFT expressed in degrees, using numpy.angle	$y_1 = \angle\,\mathrm{FFT}\left(y_0\right)$
Power spectral density (PSD)	Optional: output in decibels (dB). PSD is estimated using Welch's method (see scipy.signal.welch)	$y_1 = \mathrm{PSD}\left(y_0\right)$ or $y_1 = 10\log_{10}\left(\mathrm{PSD}\left(y_0\right)\right)$ (dB)

Note: FFT and inverse FFT are performed using frequency shifting if the option is enabled in DataLab settings (see *Settings*).

Frequency filters

Create a new signal which is the result of applying a frequency filter to each selected signal.

The following filters are available:

Filter	Description
Low-pass	Filter out high frequencies, above a cutoff frequency
High-pass	Filter out low frequencies, below a cutoff frequency
Band-pass	Filter out frequencies outside a range
Band-stop	Filter out frequencies inside a range

For each filter, the following methods are available:

Method	Description
Bessel	Bessel filter, using SciPy's scipy.signal.bessel function
Brick wall	Ideal (rectangular) filter in the frequency domain
Butterworth	Butterworth filter, using SciPy's scipy.signal.butter function
Chebyshev I	Chebyshev type I filter, using SciPy's scipy.signal.cheby1 function
Chebyshev II	Chebyshev type II filter, using SciPy's scipy.signal.cheby2 function
Elliptic	Elliptic filter, using SciPy's scipy.signal.ellip function

Fitting

Fit a model to each selected signal. This can be done automatically or through an interactive curve fitting dialog.

Model	Equation
Linear	$y = c_0 + c_1 \cdot x$
Polynomial	$y = c_0 + c_1 \cdot x + c_2 \cdot x^2 + ... + c_n \cdot x^n$
Gaussian	$y = y_0 + \frac{A}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma}\right)^2\right)$
Lorentzian	$y = y_0 + \frac{A}{\pi\sigma} \cdot \frac{1}{1 + \left(\frac{x - x_0}{\sigma}\right)^2}$
Voigt	$y = y_0 + A \cdot \frac{\Re\left(\exp(-z^2) \cdot \mathrm{erfc}(-j \cdot z)\right)}{\sqrt{2\pi}\sigma}$ with $z = \frac{x - x_0 - j \cdot \sigma}{\sqrt{2}\sigma}$
Multi-Gaussian	$y = y_0 + \sum_{i=0}^{N} \frac{A_i}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{1}{2}\left(\frac{x - x_{0,i}}{\sigma_i}\right)^2\right)$
Multi-Lorentzian	$y = y_0 + \sum_{i=0}^{N} \frac{A_i}{\pi\sigma_i} \cdot \frac{1}{1 + \left(\frac{x - x_{0,i}}{\sigma_i}\right)^2}$
Planck	$y = y_0 + A \cdot \frac{2h \cdot c^2}{\lambda^5} \cdot \left(\exp\left(\frac{h \cdot c}{\lambda \cdot k \cdot T}\right) - 1\right)^{-1}$
Two half Gaussians	$y = y_0 + A \cdot \frac{1}{\sqrt{2\pi}\sigma_0} \cdot \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma_0}\right)^2\right)$ if $x < x_0$ $y = y_0 + A \cdot \frac{1}{\sqrt{2\pi}\sigma_1} \cdot \exp\left(-\frac{1}{2}\left(\frac{x - x_1}{\sigma_1}\right)^2\right)$ otherwise
Two back-to-back exponentials	$y = y_0 + A_1 \cdot \exp\left(-x/\tau_1\right)$ if $x < 0$ $y = y_0 + A_2 \cdot \exp(-x/\tau_2)$ otherwise
Exponential	$y = y_0 + A \exp\left(B \cdot x\right)$
Sinusoidal	$y = y_0 + A \sin\left(2\pi f \cdot x + \phi\right)$
Cumulative Distribution Function (CDF)	$y = y_0 + A\,\mathrm{erf}\left(\frac{x - x_0}{\sqrt{2}\sigma}\right)$

Windowing

Create a new signal which is the result of applying a window function to each selected signal.

The following window functions are available:

Window function	Reference
Barthann	`scipy.signal.windows.barthann()`
Bartlett	`numpy.bartlett()`
Blackman	`scipy.signal.windows.blackman()`
Blackman-Harris	`scipy.signal.windows.blackmanharris()`
Bohman	`scipy.signal.windows.bohman()`
Boxcar (rectangular)	`scipy.signal.windows.boxcar()`
Cosine	`scipy.signal.windows.cosine()`
Exponential	`scipy.signal.windows.exponential()`
Flat top	`scipy.signal.windows.flattop()`
Gaussian	`scipy.signal.windows.gaussian()`
Hamming	`numpy.hamming()`
Hann	`numpy.hanning()`
Kaiser	`scipy.signal.windows.kaiser()`
Lanczos	`scipy.signal.windows.lanczos()`
Nuttall	`scipy.signal.windows.nuttall()`
Parzen	`scipy.signal.windows.parzen()`
Taylor	`scipy.signal.windows.taylor()`
Tukey	`scipy.signal.windows.tukey()`

Detrending

Create a new signal which is the detrending of each selected signal. This features is based on SciPy's scipy.signal.detrend function.

The following parameters are available:

Parameter	Description
Method	Detrending method: 'linear' or 'constant'. See SciPy's scipy.signal.detrend function.

Interpolation

Create a new signal which is the interpolation of each selected signal with respect to a second signal X-axis (which might be the same as one of the selected signals).

The following interpolation methods are available:

Method	Description
Linear	Linear interpolation, using using NumPy's interp function
Spline	Cubic spline interpolation, using using SciPy's scipy.interpolate.splev function
Quadratic	Quadratic interpolation, using using NumPy's polyval function
Cubic	Cubic interpolation, using using SciPy's Akima1DInterpolator class
Barycentric	Barycentric interpolation, using using SciPy's BarycentricInterpolator class
PCHIP	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) interpolation, using using SciPy's PchipInterpolator class

Resampling

Create a new signal which is the resampling of each selected signal.

The following parameters are available:

Parameter	Description
Method	Interpolation method (see previous section)
Fill value	Interpolation fill value (see previous section)
Xmin	Minimum X value
Xmax	Maximum X value
Mode	Resampling mode: step size or number of points
Step size	Resampling step size
Number of points	Resampling number of points

Stability analysis

Create a new signal which is the result of a stability analysis of each selected signal.

The following stability analysis methods are available:

Function	Description
Allan variance	Measure of the stability of a signal: defined as the variance of the difference between two successive measurements as a function of the time interval between them.
Allan deviation	Square root of the Allan variance.
Overlapping Allan deviation	A more robust version of the Allan variance that overlaps successive segments to improve statistical confidence.
Modified Allan variance	A variation of the Allan variance that accounts for phase noise by introducing a filtering operation.
Hadamard variance	An alternative to Allan variance, more robust to linear frequency drift in the signal
Total variance	Extends the Allan variance concept to cover all possible averaging intervals.
Time deviation	Derived from Allan deviation, quantifies stability in terms of time rather than frequency.

Note: The "All stability features" option allows to compute all stability analysis methods at once.

X-Y Mode

Simulate the X-Y mode of an oscilloscope.

2.2.7 Analysis features on Signals

This section describes the signal analysis features available in DataLab.

See also:

Operations on Signals for more information on operations that can be performed on signals, or *Processing Signals* for information on processing features on signals.

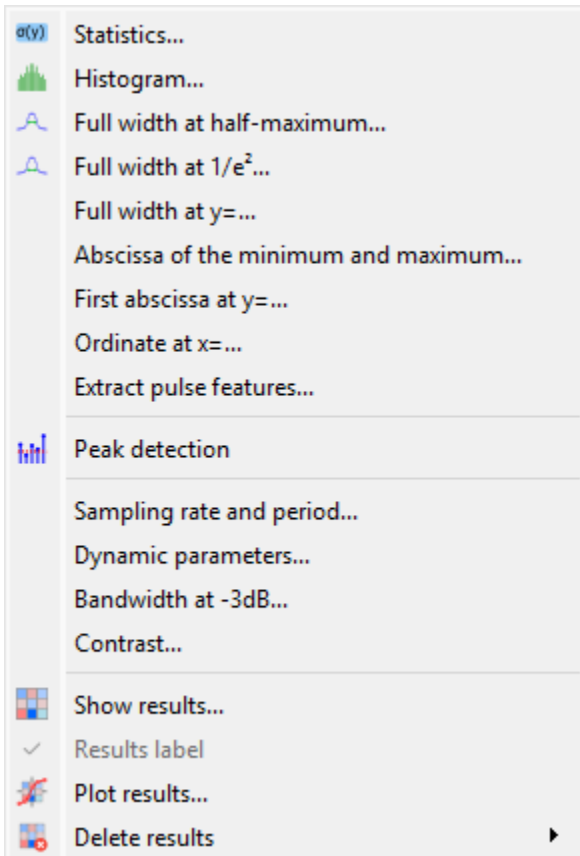


Fig. 28: Screenshot of the "Analysis" menu.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "Analysis" menu allows you to perform various computations on the selected signals, such as statistics, full width at half-maximum, or full width at $1/e^2$.

Note: In DataLab vocabulary, an "analysis" is a feature that computes a scalar result from a signal. This result is stored as metadata, and thus attached to signal. This is different from a "processing" which creates a new signal from an existing one.

Statistics

Compute statistics on selected signal and show a summary table.

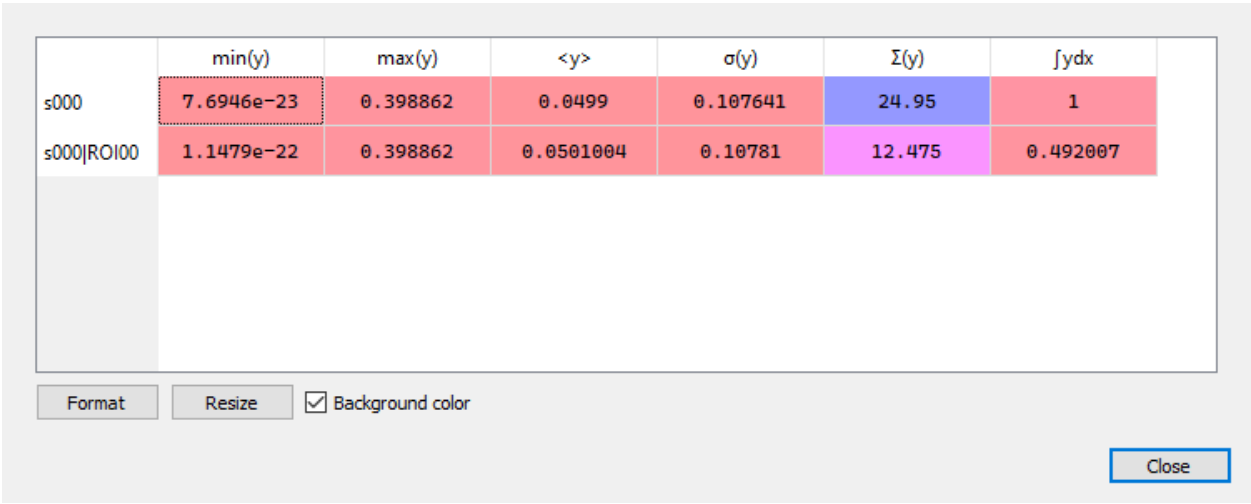


Fig. 29: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

Histogram

Compute histogram of selected signal and show it.

Parameters are:

Parameter	Description
Bins	Number of bins
Lower limit	Lower limit of the histogram
Upper limit	Upper limit of the histogram

Full width at half-maximum

Compute the Full Width at Half-Maximum (FWHM) of selected signal, using one of the following methods:

Method	Description
Zero-crossing	Find the zero-crossings of the signal after having centered its amplitude around zero
Gauss	Fit data to a Gaussian model using least-square method
Lorentz	Fit data to a Lorentzian model using least-square method
Voigt	Fit data to a Voigt model using least-square method

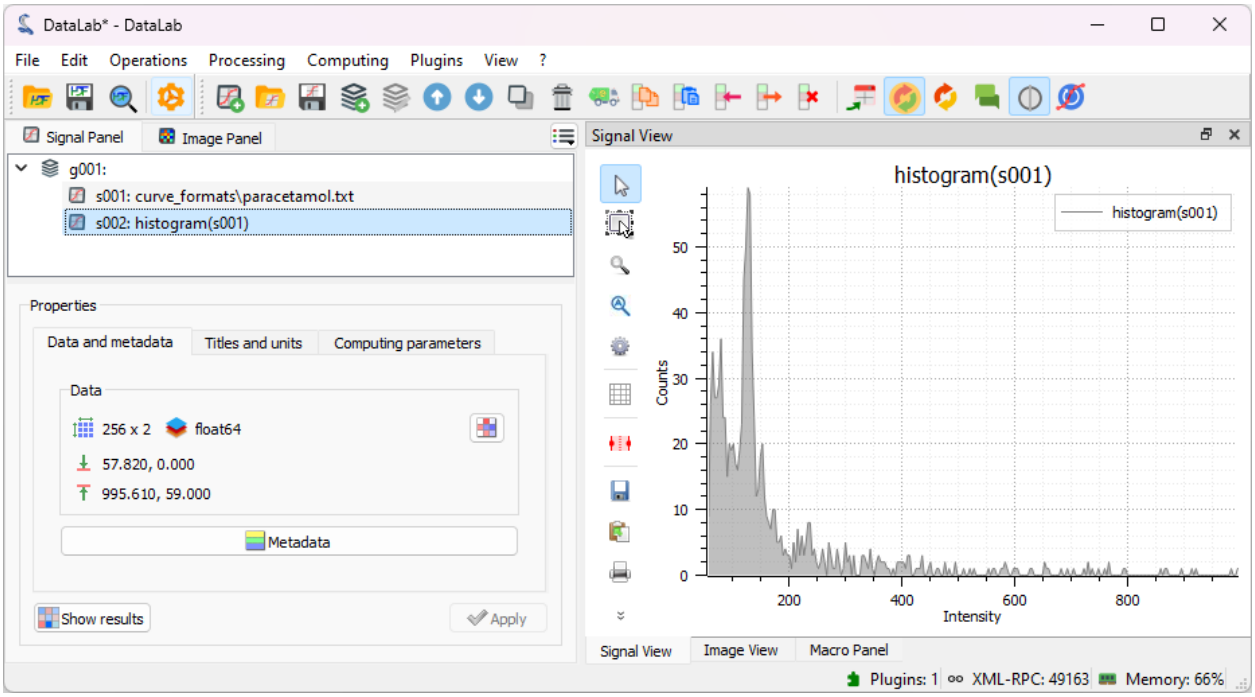


Fig. 30: Example of histogram.

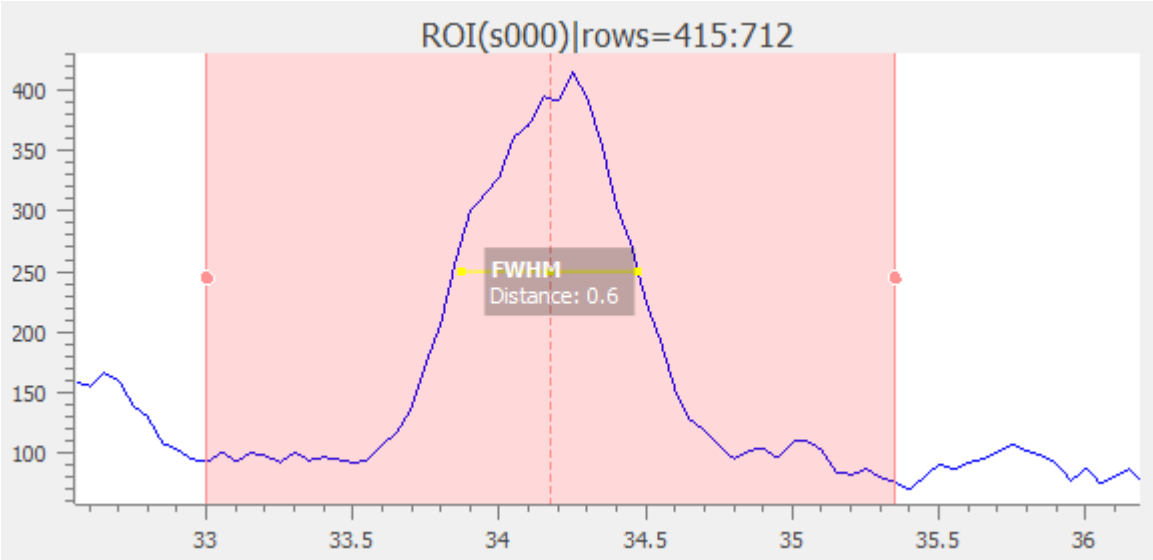


Fig. 31: The computed result is displayed as an annotated segment.

Full width at $1/e^2$

Fit data to a Gaussian model using least-square method. Then, compute the full width at $1/e^2$.

Note: Computed scalar results are systematically stored as metadata. Metadata is attached to signal and serialized with it when exporting current session in a HDF5 file.

Arguments of the min and max

Compute the smallest argument of the minima and the smallest argument of the maxima of the selected signal.

Abscissa at y=...

Compute the abscissa at a given ordinate value for the selected signal. If there is no solution, the displayed result is NaN. If there are multiple solutions, the displayed result is the smallest value.

Peak detection

Create a new signal from semi-automatic peak detection of each selected signal.

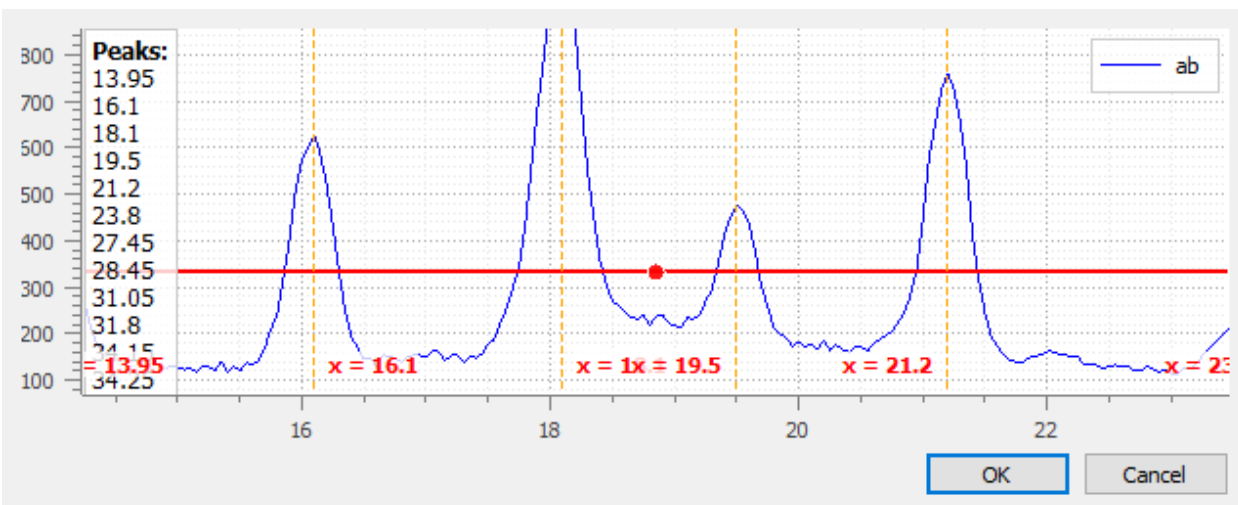


Fig. 32: Peak detection dialog: threshold is adjustable by moving the horizontal marker, peaks are detected automatically (see vertical markers with labels indicating peak position)

Sampling rate and period

Compute the sampling rate and period of selected signal.

> **Warning:** This feature assumes that the X values are regularly spaced.

Dynamic parameters

Compute the following dynamic parameters on selected signal:

Parameter	Description
f	Frequency (sinusoidal fit)
ENOB	Effective Number Of Bits
SNR	Signal-to-Noise Ratio
SINAD	Signal-to-Noise And Distortion Ratio
THD	Total Harmonic Distortion
SFDR	Spurious-Free Dynamic Range

Bandwidth at -3 dB

Determine the bandwidth at -3 dB by identifying the range of abscissa values where the signal remains greater than its maximum value minus 3 dB.

> **Warning:** This feature requires that the signal is expressed in decibels (dB).

Contrast

Compute the contrast of selected signal.

The contrast is defined as the ratio of the difference and the sum of the maximum and minimum values:

$$\text{Contrast} = \frac{\max(y) - \min(y)}{\max(y) + \min(y)}$$

> **Note:** This feature assumes that the signal is a profile from an image, where the contrast is meaningful. This justifies the optical definition of contrast.

Extract pulse features

Perform comprehensive pulse analysis on selected signals, automatically extracting timing and amplitude characteristics for step and square pulse signals.

This feature provides automated pulse characterization with intelligent signal type recognition and robust parameter extraction for digital signal analysis, oscilloscope measurements, and pulse timing validation.

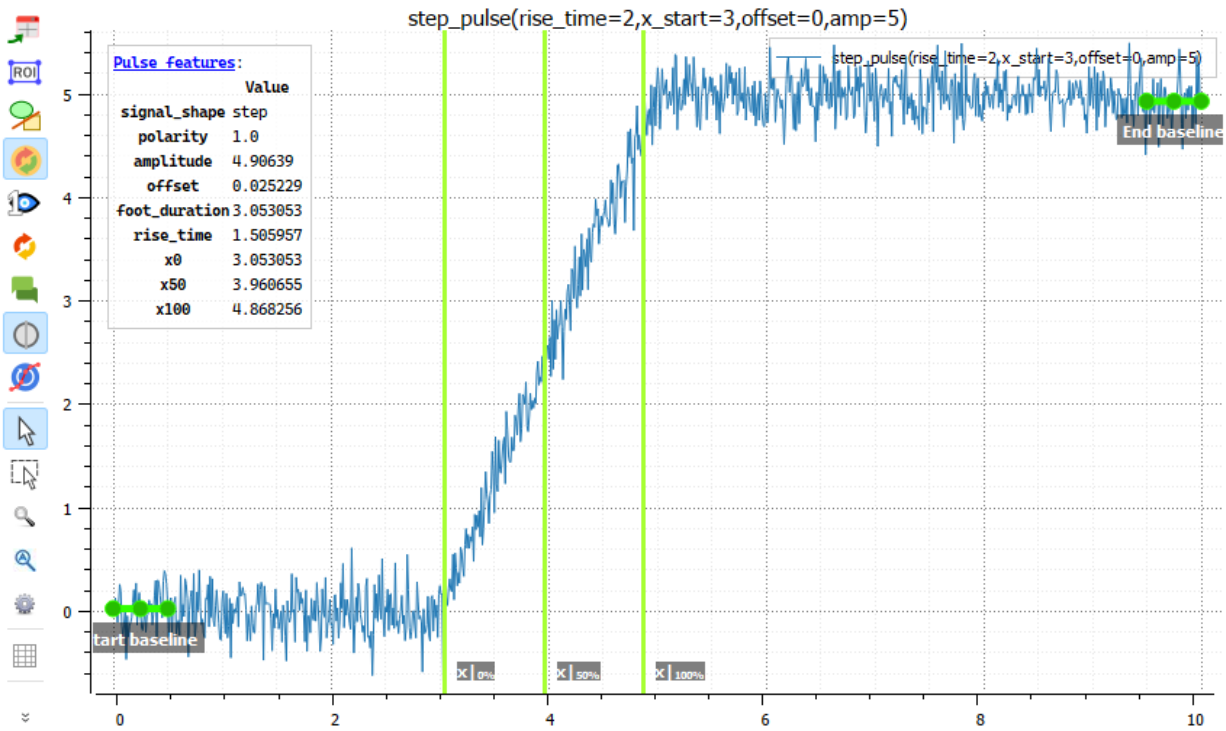


Fig. 33: Pulse features results as displayed in the Signal View.

Key Capabilities:

- **Automated signal recognition**: Heuristically identifies signal type (step, square, or other) for optimal analysis
- **Polarity detection**: Automatically determines positive/negative pulse polarity using baseline comparison
- **Comprehensive timing measurements**: Extracts rise time, fall time, FWHM, and timing parameters at specific amplitude fractions
- **Baseline characterization**: Analyzes start and end baseline regions for accurate feature computation
- **Robust algorithms**: Uses advanced statistical methods with noise tolerance and error handling

Parameters:

Parameter	Description
Signal shape	Signal type selection: Auto (automatic detection), Step, or Square
Start baseline min	Lower X boundary for the start (initial) baseline region
Start baseline max	Upper X boundary for the start (initial) baseline region
End baseline min	Lower X boundary for the end (final) baseline region
End baseline max	Upper X boundary for the end (final) baseline region
Rise/Fall time	Reference levels for rise/fall time measurement with predefined choices: 5%-95% (High precision), 10%-90% (IEEE standard), 20%-80% (Noisy signals), 25%-75% (Alternative)

Extracted Features:

The analysis computes the following pulse characteristics:

Feature	Description
Signal shape	Detected signal type (STEP, SQUARE, or OTHER)
Polarity	Signal polarity (+1 for positive, -1 for negative pulses)
Amplitude	Peak-to-peak amplitude of the pulse
Offset	DC offset (baseline level)
Rise time	Time from the lower to upper reference levels during rising edge
Fall time	Time from the lower to upper reference levels during falling edge (square pulses only)
FWHM	Full Width at Half Maximum (square pulses only)
x50	Time at which signal reaches 50% of maximum amplitude
x100	Time at which signal reaches 100% of maximum amplitude (plateau start)
Foot duration	Duration of flat region before pulse rise
Baseline ranges	Extracted start and end baseline boundary coordinates

Note: Results are displayed in a comprehensive table showing all extracted parameters, and as visual annotations on the signal plot. For step signals, fall-related parameters (fall_time, fwhm) are not applicable and show as None. The feature works best with well-defined pulse signals that have clear baseline regions.

Show results

Show the results of all analyses performed on the selected signals. This shows the same table as the one shown after having performed a computation.

Results label

Toggle the visibility of result labels on the plot. When enabled, this checkable menu item displays result annotations (such as FWHM, peak positions, or other analysis markers) directly on the signal plot.

This option is synchronized between Signal and Image panels and persists across sessions. It is only enabled when results are available for the selected signal.

Plot results

Plot the results of analyses performed on the selected signals, with user-defined X and Y axes (e.g. plot the FWHM as a function of the signal index).

2.2.8 View options for Signals

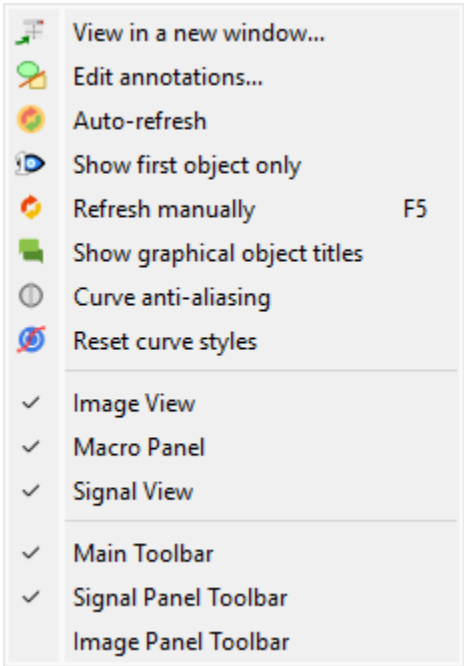


Fig. 34: Screenshot of the "View" menu.

When the "Signal Panel" is selected, the menus and toolbars are updated to provide signal-related actions.

The "View" menu allows you to visualize the current signal or group of signals. It also allows you to show/hide titles, to enable/disable anti-aliasing, or to refresh the visualization.

View in a new window

Open a new window to visualize the selected signals.

This option allows you to visualize the selected signals in a separate window, in which you can visualize the data more comfortably (e.g., by maximizing the window) and you can also annotate the data.

When you click on the button "Annotations" in the toolbar of the new window, the annotation editing mode is activated (see the section "Edit annotations" below).

Edit annotations

Open a new window to edit annotations.

This option allows you to show the selected signals in a separate window, in which you can visualize the data more comfortably (e.g., by maximizing the window) as well as to add or edit annotations.

Annotations are used to add comments or labels to the data, and they can be used to highlight specific events or regions of interest.

Note: The annotations are saved in the metadata of the signal, so they are persistent and will be displayed every time you visualize the signal.

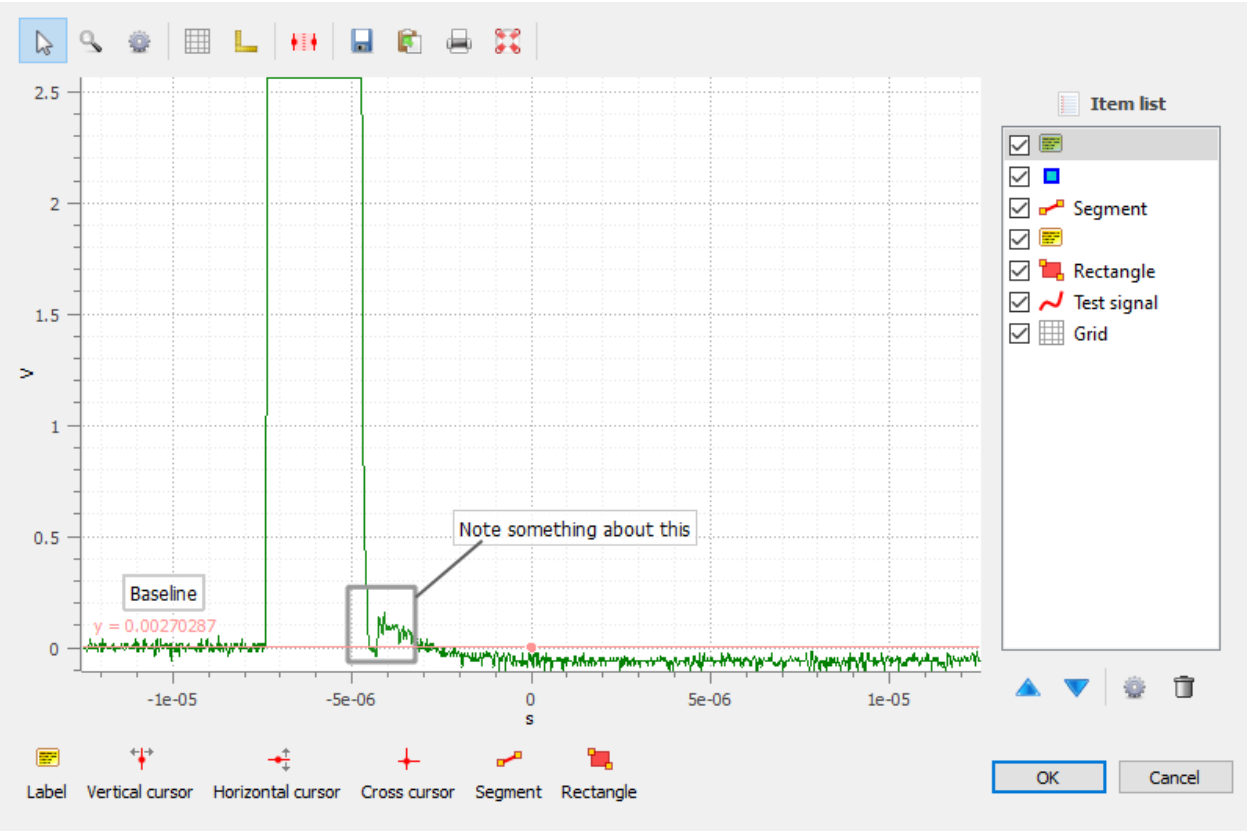


Fig. 35: Annotations may be added in the separate view.

The typical workflow to edit annotations is as follows:

1. Select the "Edit annotations" option.
2. In the new window, add annotations by clicking on the corresponding buttons at the bottom of the window.
3. Eventually, customize the annotations by changing their properties (e.g., the text, the color, the position, etc.) using the "Parameters" option in the context menu of the annotations.
4. When you are done, click on the "OK" button to save the annotations. This will close the window and the annotations will be saved in the metadata of the signal and will be displayed in the main window.

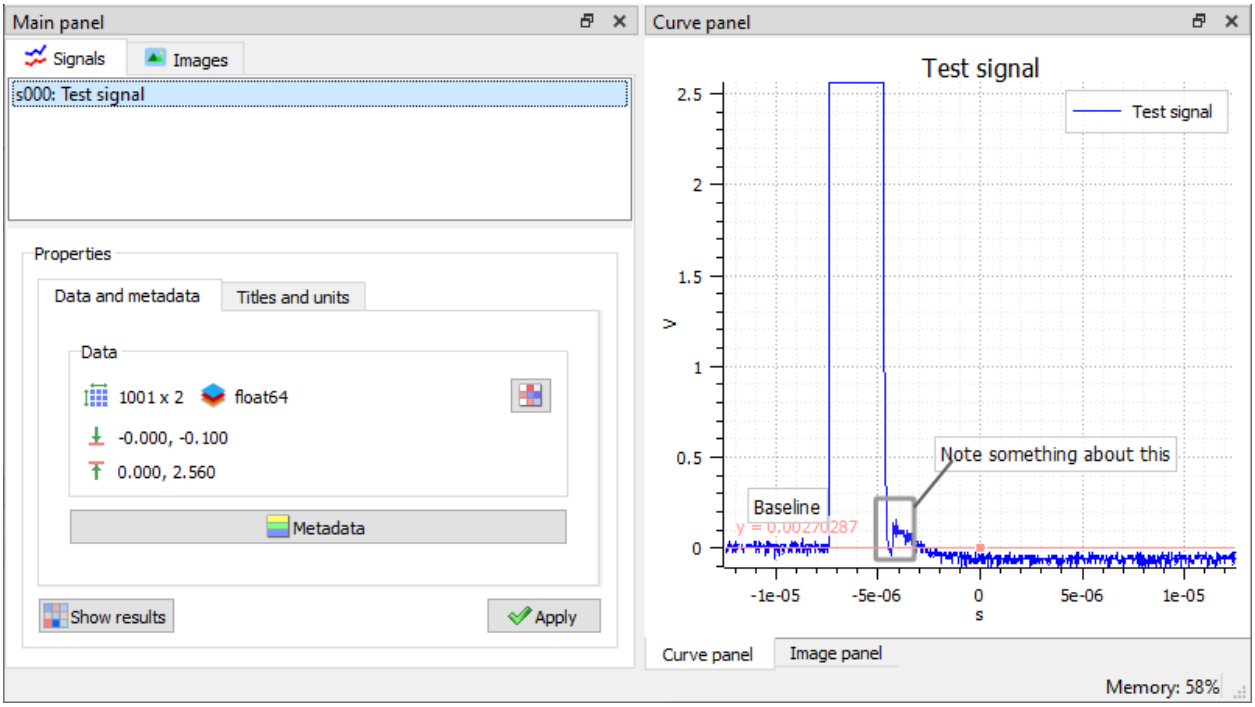


Fig. 36: Annotations are now part of the signal metadata.

Note: Annotations may be copied from a signal to another by using the "copy/paste metadata" features.

Auto-refresh

Automatically refresh the visualization when the data changes:

- When enabled (default), the plot view is automatically refreshed when the data changes.
- When disabled, the plot view is not refreshed until you manually refresh it by using the "Refresh manually" menu entry.

Even though the refresh algorithm is optimized, it may still take some time to refresh the plot view when the data changes, especially when the data set is large. Therefore, you may want to disable the auto-refresh feature when you are working with large data sets, and enable it again when you are done. This will avoid unnecessary refreshes.

Show only first selected signal

Toggle between showing all selected signals or only the first one.

When this option is enabled, only the first selected signal is displayed in the plot view. This may be useful when multiple signals are selected and focusing on a single signal is (temporarily) preferred.

Refresh manually

Refresh the visualization manually.

This triggers a refresh of the plot view. It is useful when the auto-refresh feature is disabled, or when you want to force a refresh of the plot view.

Show graphical object titles

Show/hide titles of analysis results or annotations.

This option allows you to show or hide the titles of the graphical objects (e.g., the titles of the analysis results or annotations). Hiding the titles can be useful when you want to visualize the data without any distractions, or if there are too many titles and they are overlapping.

Curve anti-aliasing

Enable/disable anti-aliasing of curves.

Anti-aliasing makes the curves look smoother, but it may also make them look less sharp.

Note: Anti-aliasing is enabled by default.

Warning: Anti-aliasing may slow down the visualization, especially when working with large data sets.

Reset curve styles

Reset the curve style cycle to the beginning.

When plotting curves, DataLab automatically assigns a color and a line style to each curve. Both parameters are chosen from a predefined list of colors and line styles, and are assigned in a round-robin fashion.

This menu entry allows you to reset the curve styles, so that the next time you plot curves, the first curve will be assigned the first color and the first line style of the predefined lists, and the loop will start again from there.

2.3 Image processing

This section describes the features specific to the image processing panel. The image processing panel can be selected by clicking on the "Images" tab at the bottom-right of the DataLab main window.

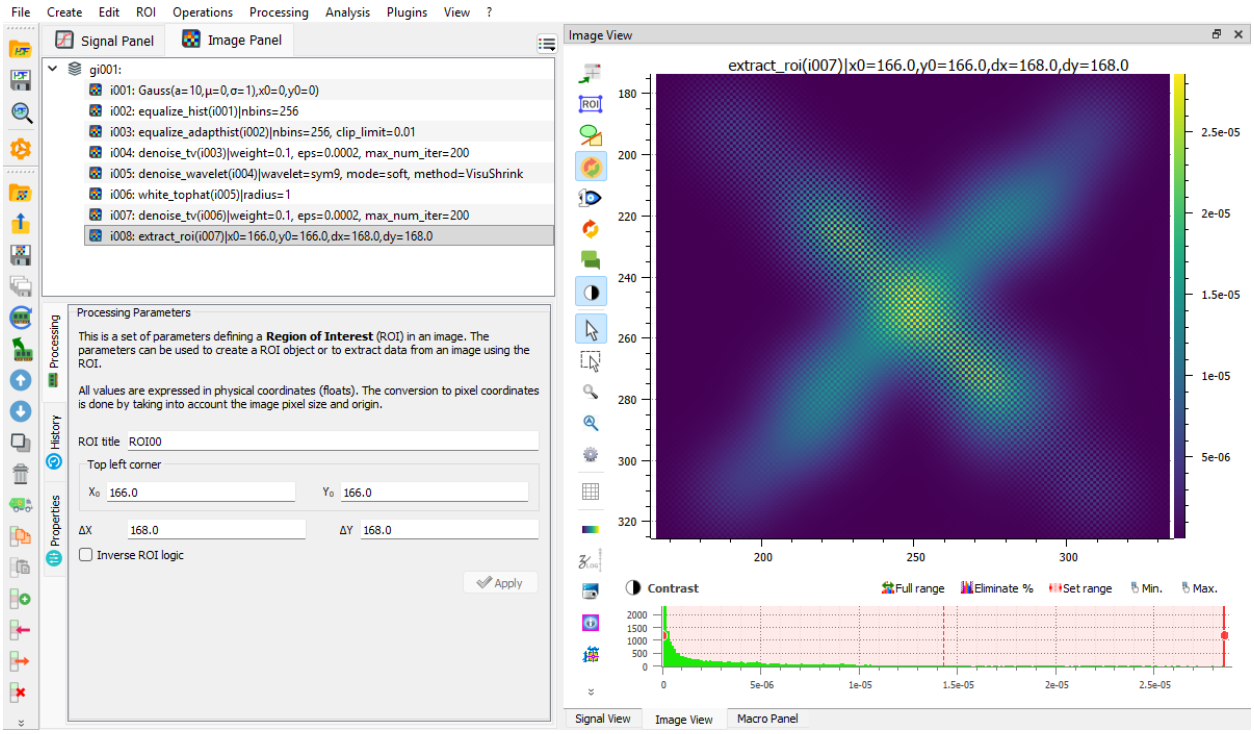


Fig. 37: DataLab main window: Image processing view

2.3.1 Open and save Images

This section describes how to open and save images (and workspaces).

Note: For creating new images from mathematical models, see *Create Images*.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "File" menu allows you to:

- Open, save and close images (see below).
- Save and restore the current workspace or browse HDF5 files (see *Overview*).
- Edit DataLab preferences (see *Settings*).

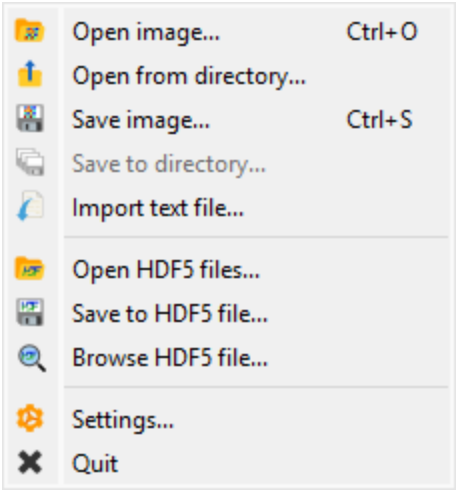


Fig. 38: Screenshot of the "File" menu.

Open image

Create a new image from the following supported filetypes:

File type	Extensions
PNG files	.png
TIFF files	.tif, .tiff
8-bit images	.jpg, .gif
NumPy arrays	.npy
MAT-Files	.mat
Text files	.txt, .csv, .asc
Andor SIF files	.sif
Princeton Instruments SPE files	.spe
Opticks GEL files	.gel
Hammamatsu NDPI files	.ndpi
PCO Camera REC files	.rec
SPIRICON files	.scor-data
FXD files	.fxd
Bitmap images	.bmp
FT-Lab files	.ima

Note: DataLab also supports any image format that can be read by the *imageio* library, provided that the associated plugin(s) are installed (see imageio documentation) and that the output NumPy array data type and shape are supported by DataLab.

To add a new file format, you may use the *imageio_formats* entry of DataLab configuration file. This entry is a formatted like the *IMAGEIO_FORMATS* object which represents the natively supported formats:

`sigima.config.IMAGEIO_FORMATS = (('*.gel', 'Opticks GEL'), ('*.spe', 'Princeton Instruments SPE'), ('*.ndpi', 'Hamamatsu Slide Scanner NDPI'), ('*.rec', 'PCO Camera REC'))`

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

Open from directory

Open multiple images from a specified directory.

Save image

Save current image (see "Open image" supported filetypes).

Save images to directory

Save all selected images to a specified directory, with configurable filename pattern and image format.

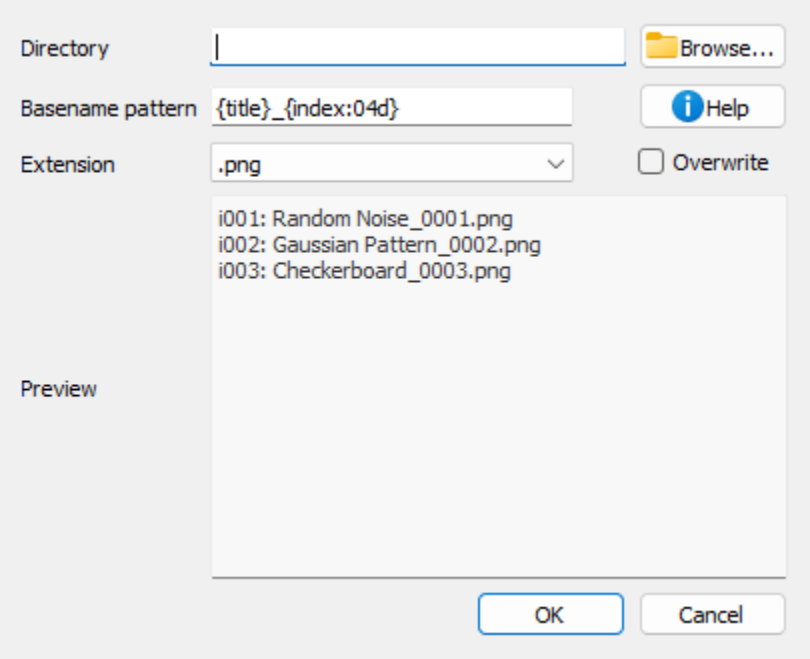


Fig. 39: Save images to directory dialog.

When you select "Save to directory…" from the File menu, a dialog appears where you can:

- **Directory**: Choose the target directory where images will be saved
- **Filename pattern**: Define a pattern for the filenames using Python format strings
- **File extension**: Select the output format (.png, .tiff, .h5ima, etc.)
- **Overwrite**: Choose whether to overwrite existing files
- **Preview**: See the list of files that will be created (with object IDs)

The filename pattern supports the following placeholders:

- `{title}`: Image title
- `{index}`: 1-based index of the image in the selection (with zero-padding)
- `{count}`: Total number of selected images
- `{xlabel}`, `{xunit}`, `{ylabel}`, `{yunit}`, `{zlabel}`, `{zunit}`: Axis labels and units
- `{metadata[key]}`: Access metadata values

You can also use format modifiers, for example `{index:03d}` will format the index with 3 digits zero-padding (001, 002, 003, etc.).

Import text file

DataLab can natively import many types of image files (e.g. TIFF, JPEG, PNG, etc.). However some specific text file formats may not be supported. In this case, you can use the *Import text file* feature, which allows you to import a text file and convert it to an image.

This feature is accessible from the *File* menu, under the *Import text file* option.

It opens an import wizard that guides you through the process of importing the text file.

Step 1: Select the source

The first step is to select the source of the text file. You can either select a file from your computer or the clipboard if you have copied the text from another application.

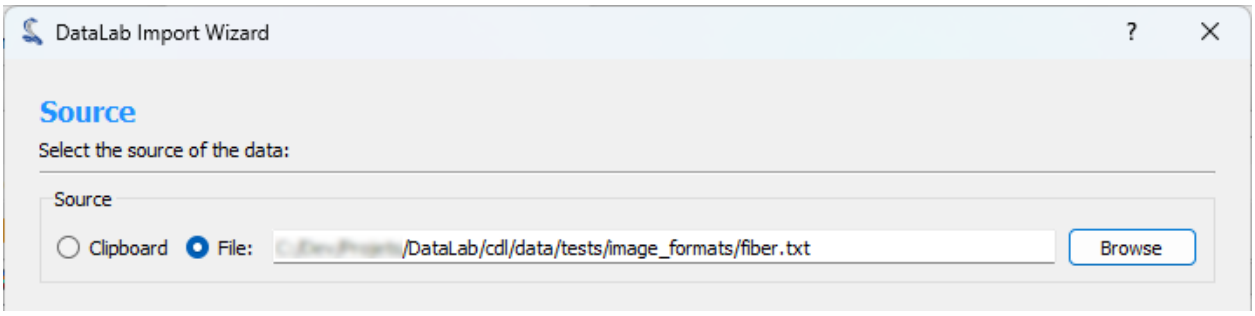


Fig. 40: Step 1: Select the source

Step 2: Preview and configure the import

The second step consists of configuring the import and previewing the result. You can configure the following options:

- **Delimiter**: The character used to separate the values in the text file.
- **Comments**: The character used to indicate that the line is a comment and should be ignored.
- **Rows to Skip**: The number of rows to skip at the beginning of the file.
- **Maximum Number of Rows**: The maximum number of rows to import. If the file contains more rows, they will be ignored.
- **Transpose**: If checked, the rows and columns will be transposed.
- **Data type**: The destination data type of the imported data.

When you are done configuring the import, click the *Apply* button to see the result.

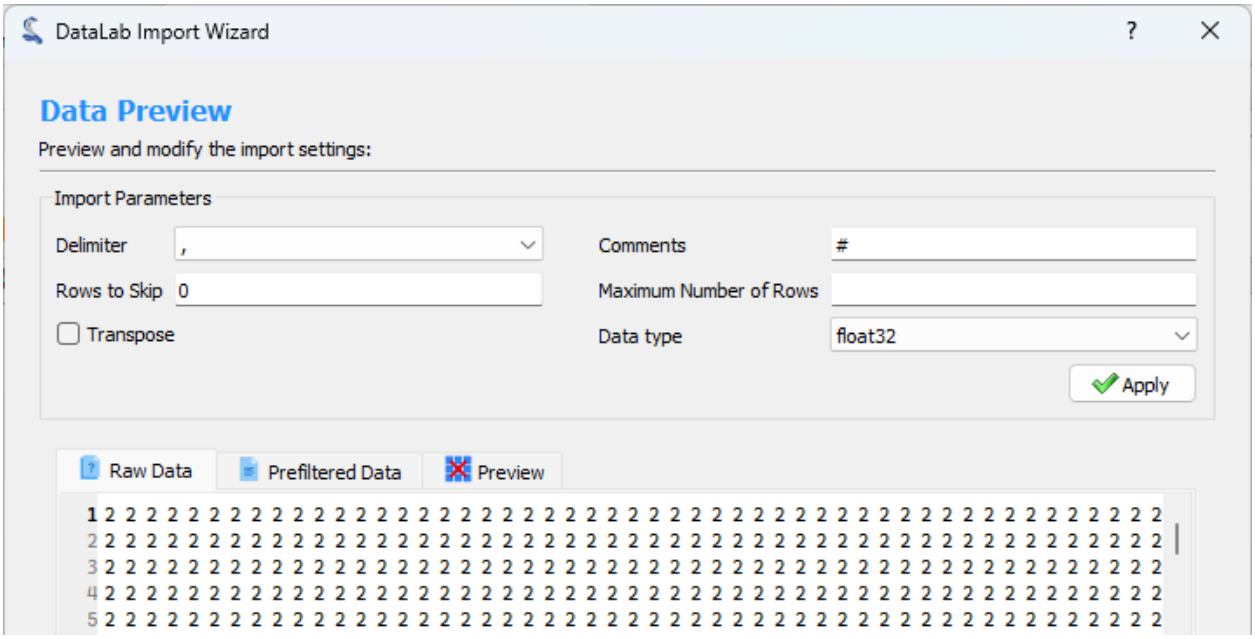


Fig. 41: Step 2: Configure the import

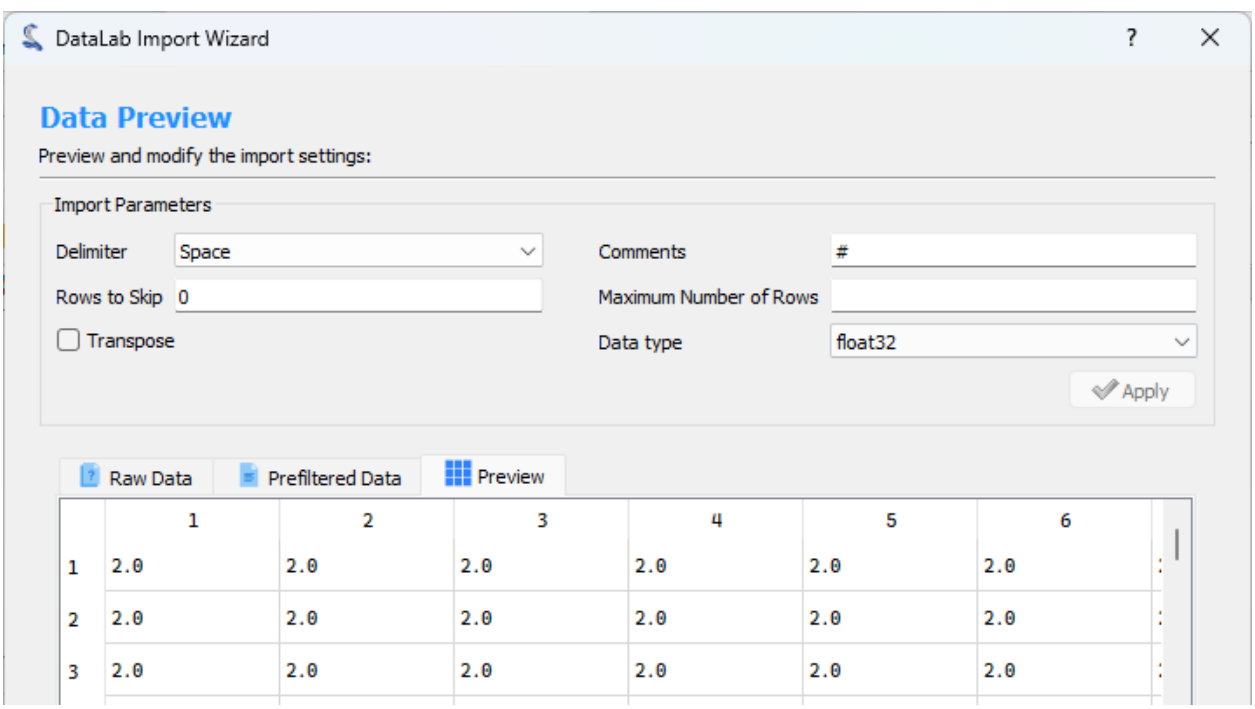


Fig. 42: Step 2: Preview the result

Step 3: Show graphical representation

The third step shows a graphical representation of the imported data. You can use the *Finish* button to import the data into DataLab workspace.

2.3.2 Create Images

This section describes how to create new images from various mathematical models.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "Create" menu allows you to create new images from various models (see below).

New image

Create a new image from various models (supported datatypes: uint8, uint16, int16, float32, float64):

Icon	Model	Equation
	Zero	$z[i] = 0$
	Normal distribution	$z[i]$ is normally distributed with configurable mean and standard deviation
	Poisson distribution	$z[i]$ is Poisson distributed with configurable mean
	Uniform distribution	$z[i]$ is uniformly distributed between two configurable bounds
	2D Gaussian	$z = A \cdot \exp\left(-\frac{\left(\sqrt{(x-x_0)^2+(y-y_0)^2}-\mu\right)^2}{2\sigma^2}\right)$
	2D Ramp	$z = A(x-x_0) + B(y-y_0) + C$
	Checkerboard	Alternating square pattern for calibration and spatial frequency analysis
	Sinusoidal grating	$z = A\sin(2\pi(f_x \cdot x + f_y \cdot y) + \varphi) + C$
	Ring pattern	Concentric circular rings for radial analysis
	Siemens star	Radial spoke pattern for resolution testing
	2D sinc	$z = A \cdot \mathrm{sinc}\left(\frac{\sqrt{(x-x_0)^2+(y-y_0)^2}}{\sigma}\right) + C$

2.3.3 Manipulate metadata and annotations

This section describes how to manipulate metadata and annotations in DataLab.

Image metadata contains various information about the image or its representation, such as view settings, Regions Of Interest (ROIs), processing chain history, analysis results, and any other information that you may have added to the metadata of an image (or that comes from the image file itself).

See also:

For more information about Regions Of Interest (ROIs), see the *Regions Of Interest (ROI)* section.

The "Edit" menu allows you to perform classic editing operations on the current image or group of images (create/rename group, move up/down, delete image/group of images, etc.).

As detailed below, it also allows you to:

- Navigate and utilize the processing chain history through actions like "Recompute" and "Select source objects".

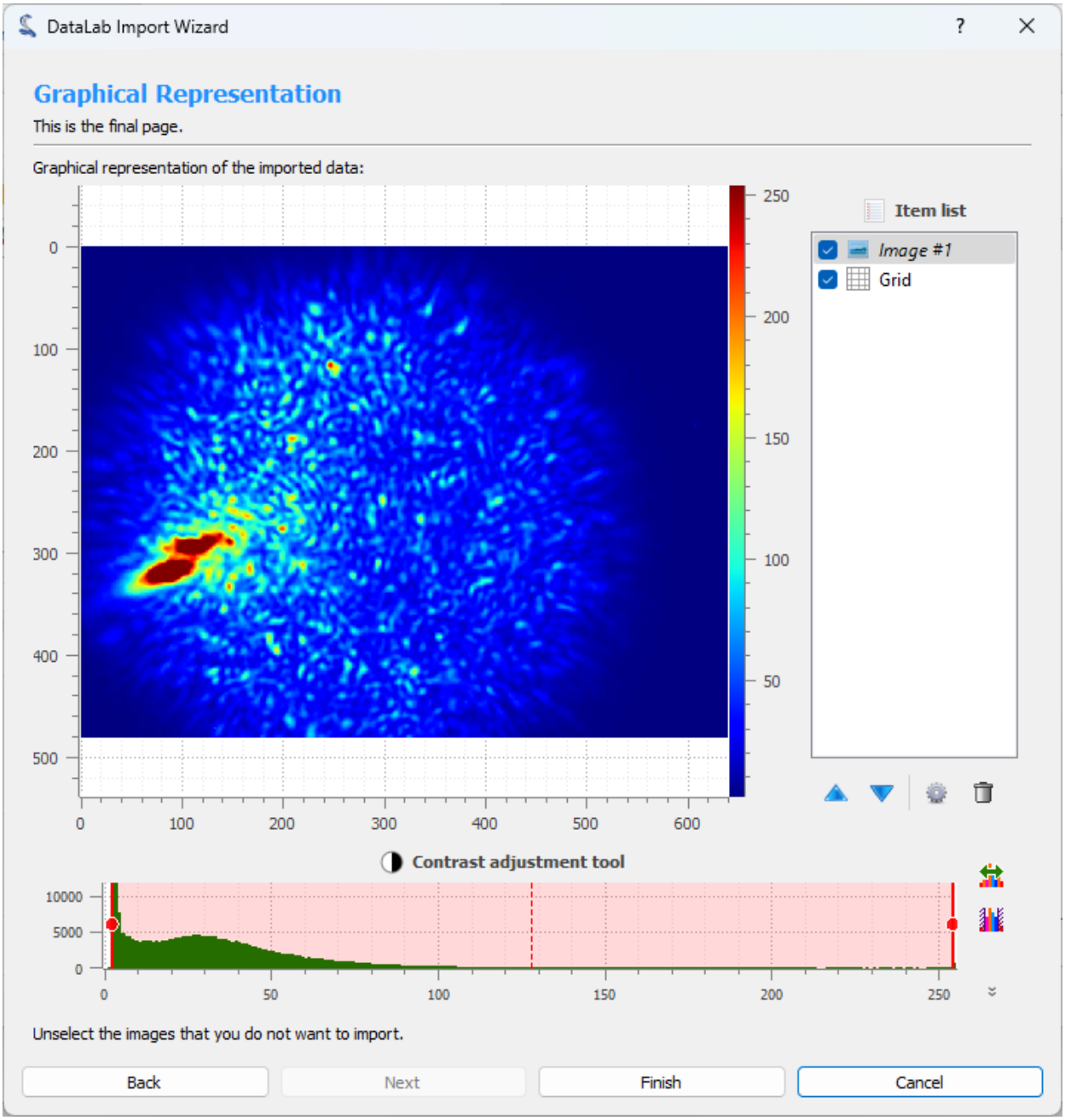


Fig. 43: Step 3: Show graphical representation

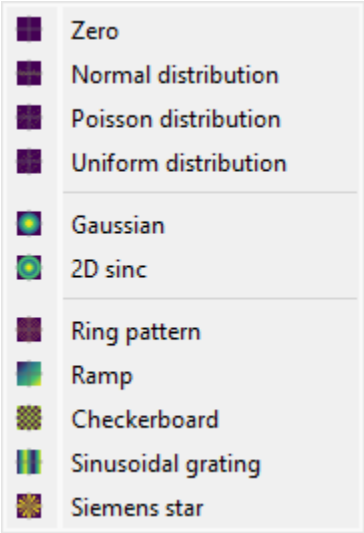


Fig. 44: Screenshot of the "Create" menu.

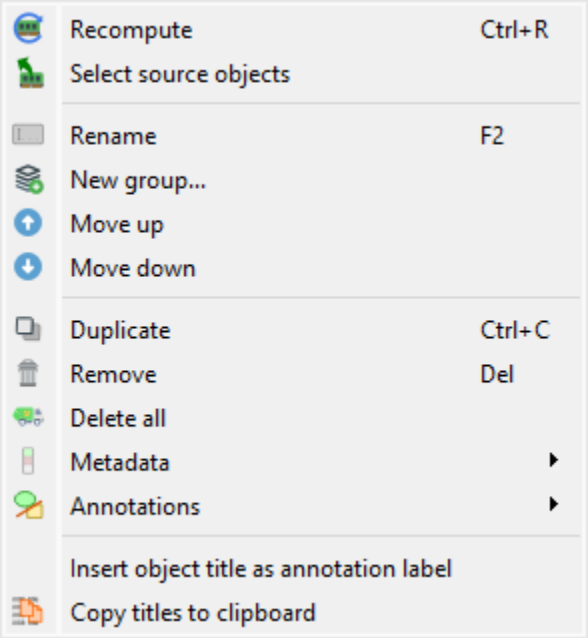


Fig. 45: Screenshot of the "Edit" menu.

- Manipulate metadata and annotations associated with the current image, thanks to the "Metadata" and "Annotations" submenus which provide the following features.

Recompute

The "Recompute" action allows you to recompute the selected image(s) using their original processing parameters. This is useful when you want to re-execute the processing chain that was used to create an image, for example after modifying global settings or dependencies.

Note: This action is only available for images that were created through processing operations and have stored processing parameters.

Select source objects

The "Select source objects" action allows you to select the source object(s) that were used to create the currently selected image. This helps trace back the processing history and understand which original images were used as input for the current result.

Note: This action is only available when exactly one image is selected and that image has source object references.

Metadata

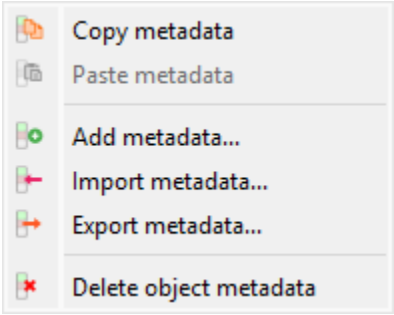


Fig. 46: Screenshot of the "Metadata" submenu.

Copy/paste metadata

As metadata contains useful information about the image, it can be copied and pasted from one image to another by selecting the "Copy metadata" and "Paste metadata" actions in the "Edit" menu.

This feature allows you to tranfer those information from one image to another:

- *Regions Of Interest (ROIs)*: that is a very efficient way to reuse the same ROI on different images and easily compare the results of the analysis on those images
- Analyze results, such as a centroid position or a contour detection (the relevance of transferring such information depends on the context and is up to the user to decide)
- Any other information that you may have added to the metadata of an image

Note: Copying metadata from an image to another will overwrite the metadata of the destination image (for the metadata keys that are common to both images) or simply add the metadata keys that are not present in the destination image.

Import/export metadata

Metadata can also be imported and exported from/to a JSON file using the "Import metadata" and "Export metadata" actions in the "Edit" menu. This is exactly the same as the copy/paste metadata feature (see above for more details on the use cases of this feature), but it allows you to save the metadata to a file and then import it back later.

Delete metadata

When deleting metadata using the "Delete metadata" action in the "Edit" menu, you will be prompted to confirm the deletion of Region of Interests (ROIs) if they are present in the metadata. After this eventual confirmation, the metadata will be deleted, meaning that analysis results, ROIs, and any other information associated with the image will be lost.

Add metadata

The "Add metadata" action allows you to add custom metadata items to one or more selected images. This is useful for tagging images with experiment IDs, sample names, processing steps, or any other custom information.

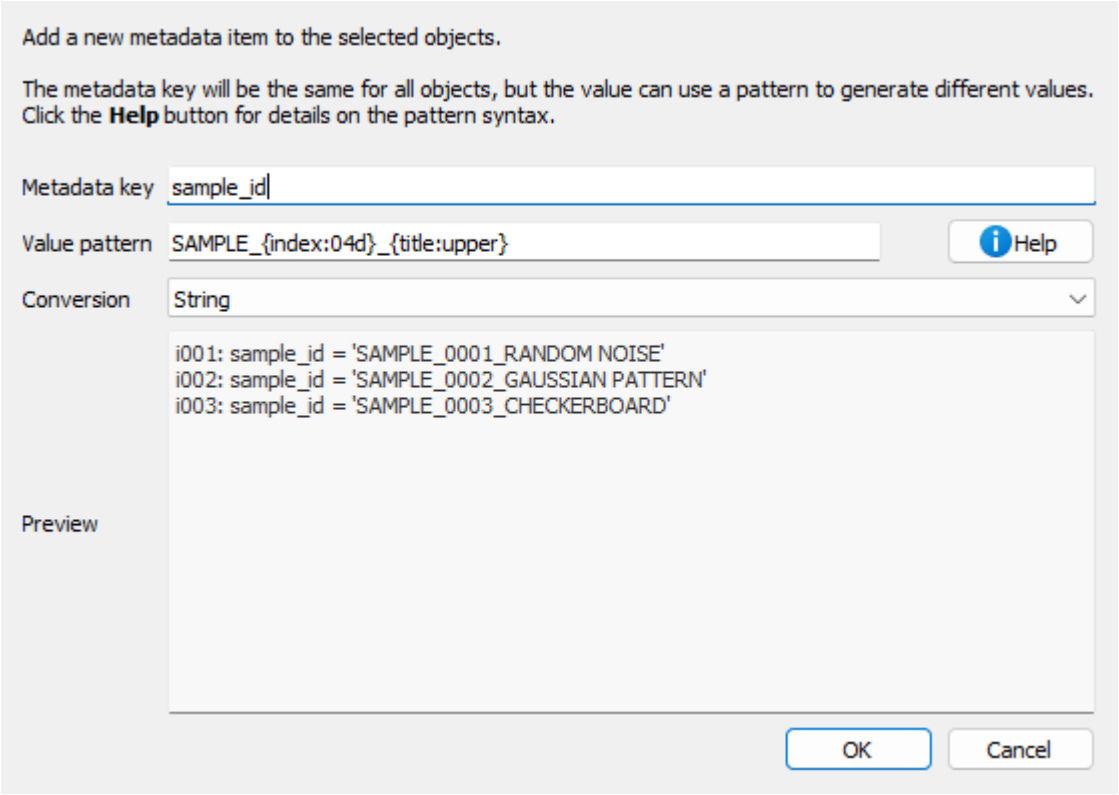


Fig. 47: Add metadata dialog.

When you select "Add metadata…" from the Edit menu, a dialog appears where you can:

- **Metadata key**: Enter the name of the metadata field to add
- **Value pattern**: Define a pattern for the metadata value using Python format strings
- **Conversion**: Choose how to store the value (string, float, integer, or boolean)
- **Preview**: See how the metadata will be added to each selected image

The value pattern supports the following placeholders:

- `{title}`: Image title
- `{index}`: 1-based index of the image in the selection
- `{count}`: Total number of selected images
- `{xlabel}`, `{xunit}`, `{ylabel}`, `{yunit}`: Axis labels and units
- `{metadata[key]}`: Access existing metadata values

You can also use format modifiers:

- `{title:upper}`: Convert to uppercase
- `{title:lower}`: Convert to lowercase
- `{index:03d}`: Format as 3-digit number with leading zeros

Examples:

- Add acquisition number: key=``acquisition``, pattern=``ACQ_{index:04d}``, conversion=string → Creates metadata like `acquisition="ACQ_0001"`
- Add exposure time: key=``exposure_ms``, pattern=``{metadata[exposure]}``, conversion=float → Copies exposure from existing metadata and converts to float
- Flag calibrated images: key=``is_calibrated``, pattern=``true``, conversion=bool → Sets `is_calibrated=True` for all selected images

Annotations

Annotations are visual elements that can be added to images to highlight specific features, mark regions of interest, or add explanatory notes. DataLab provides a dedicated submenu in the "Edit" menu for managing annotations.

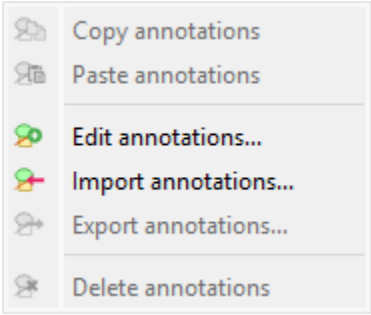


Fig. 48: Screenshot of the "Annotations" submenu.

Copy/paste annotations

Annotations can be copied from one image and pasted to one or more other images using the "Copy annotations" and "Paste annotations" actions. This is useful when you want to apply the same visual markers across multiple images.

The "Paste annotations" action is only enabled when there are annotations in the clipboard (i.e., after using "Copy annotations").

Edit annotations

The "Edit annotations" action opens a dialog where you can view, add, modify, or remove annotations from the current image. This provides a visual way to manage all annotations on an image.

Import/export annotations

Annotations can be saved to and loaded from JSON files (.dlabann extension) using the "Import annotations" and "Export annotations" actions. This allows you to:

- Save annotation sets for later reuse
- Share annotations with colleagues
- Archive annotations separately from image data
- Apply the same annotations to different images across sessions

The "Export annotations" action is only available when the selected image has annotations.

Delete annotations

The "Delete annotations" action removes all annotations from the selected image(s). This action is only enabled when the selected image(s) have annotations.

Note: Annotations are stored separately from metadata and analysis results. Deleting annotations does not affect ROIs or other metadata items.

Image titles

Image titles may be considered as metadata from a user point of view, even if they are not stored in the metadata of the image (but in an attribute of the image object).

The "Edit" menu allows you to:

- "Add object title to plot": this action will add a label on top of the image with its title, which might be useful in combination with the "Distribute on a grid" operation (see *Processing Images*) to easily identify the images.
- "Copy titles to clipboard" : this action will copy the titles of the selected images to the clipboard, which might be useful to paste them in a text editor or in a spreadsheet.

  Example of the content of the clipboard:

```
g001:
    s001: lorentz(a=1,sigma=1,mu=0,ymin=0)
    s002: derivative(s001)
    s003: wiener(s002)
g002: derivative(g001)
    s004: derivative(s001)
    s005: derivative(s002)
    s006: derivative(s003)
g003: fft(g002)
    s007: fft(s004)
    s008: fft(s005)
    s009: fft(s006)
```

2.3.4 Regions Of Interest (ROI)

This section describes how to manipulate Regions Of Interest (ROIs) for images in DataLab.

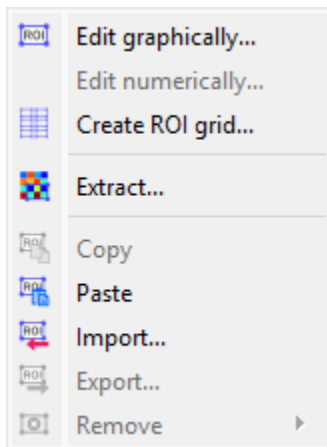


Fig. 49: Screenshot of the "ROI" menu.

The "ROI" menu allows you to manage Regions Of Interest (ROIs) associated with the current image.

See also:

For more information about image metadata, see the *Manipulate metadata and annotations* section.

The Regions Of Interest (ROI) are image areas that are defined by the user to perform specific operations, processing, or analysis on them.

ROI are taken into account almost in all computing features in DataLab:

- The "Operations" menu features are done only on the ROI if one is defined (except if the operation changes the data shape - like the resize operation - or the pixel size - like the binning operation).
- The "Processing" menu actions are performed only on the ROI if one is defined (except if the destination signal data type is different from the source's, like in the Fourier analysis features or like the thresholding operations).
- The "Analysis" menu actions are done only on the ROI if one is defined.

Note: ROI are stored as metadata, and thus attached to image.

The "ROI" menu allows you to:

- "Edit graphically" : open a dialog box to manage ROI associated with the selected image (add, remove, move, resize, etc.). The ROI definition dialog is exactly the same as ROI extraction (see below).

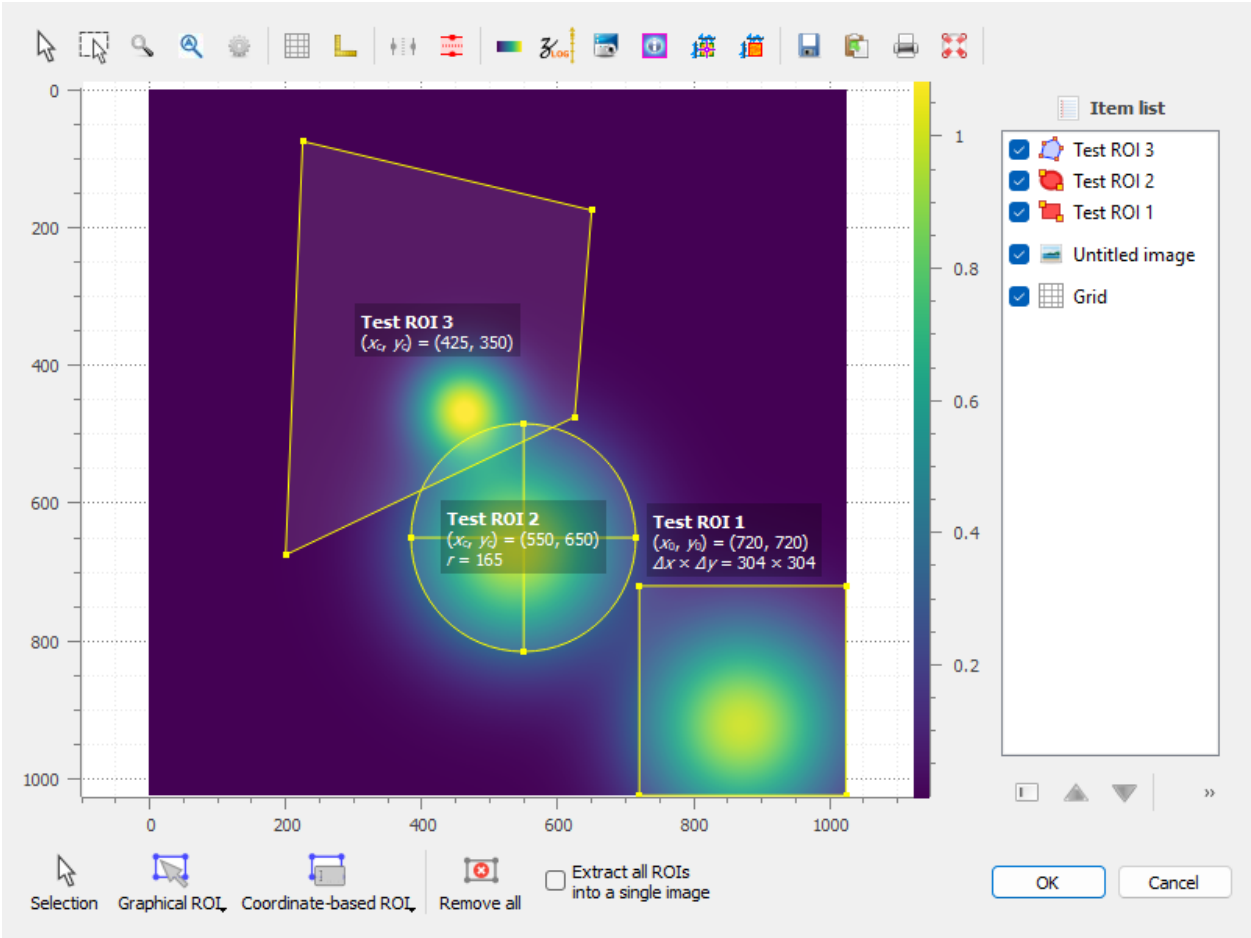


Fig. 50: An image with an ROI.

- "Edit numerically": open a dialog box to edit the parameters of the selected ROIs numerically (i.e. using a simple form). This allows you to define or modify ROIs based on numerical values.
- "Create ROI grid" : open a dialog box to create a grid of ROIs on the selected image. This allows you to define multiple ROIs distributed evenly across the image.
- "Extract" : extract the defined ROI from the selected images. This will create a new image for each ROI (or a single image, if the "Extract all ROIs into a single image" option is selected in the dialog), with the same metadata as the original image, but with the data corresponding to the ROI only. The new images will be added to the current workspace.
- "Copy" : copy the defined ROI from the selected images to the clipboard. This allows you to paste the ROI into another image or use it in other operations.
- "Paste" : paste the copied ROI from the clipboard to the selected images. This will add the ROI to the images, allowing you to define or modify ROIs based on previously copied ones.
- "Import" : import ROIs from a file. This allows you to load previously saved ROIs into the current image.

- "Export" : export the defined ROIs to a file. This allows you to save the ROIs for later use or to share them with others.
- "Remove all" : remove all defined ROI for the selected images.

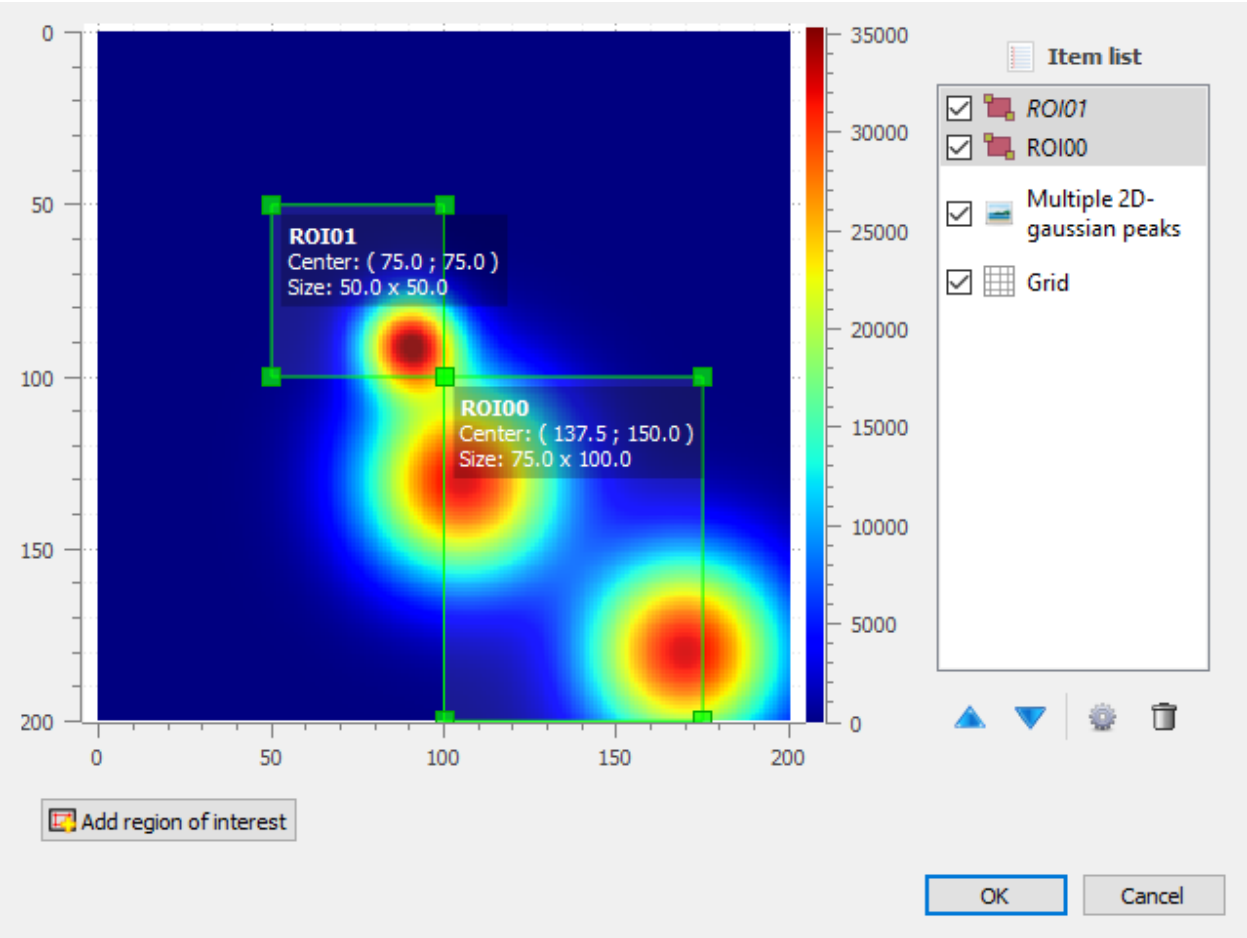


Fig. 51: ROI extraction dialog: the ROI is defined by moving the position and adjusting the size of a rectangle shape.

2.3.5 Operations on Images

This section describes the operations that can be performed on images.

See also:

Processing Images for more information on image processing features, or *Analysis features on Images* for information on analysis features on images.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "Operations" menu allows you to perform various operations on the current image or group of images. It also allows you to extract profiles, distribute images on a grid, or resize images.

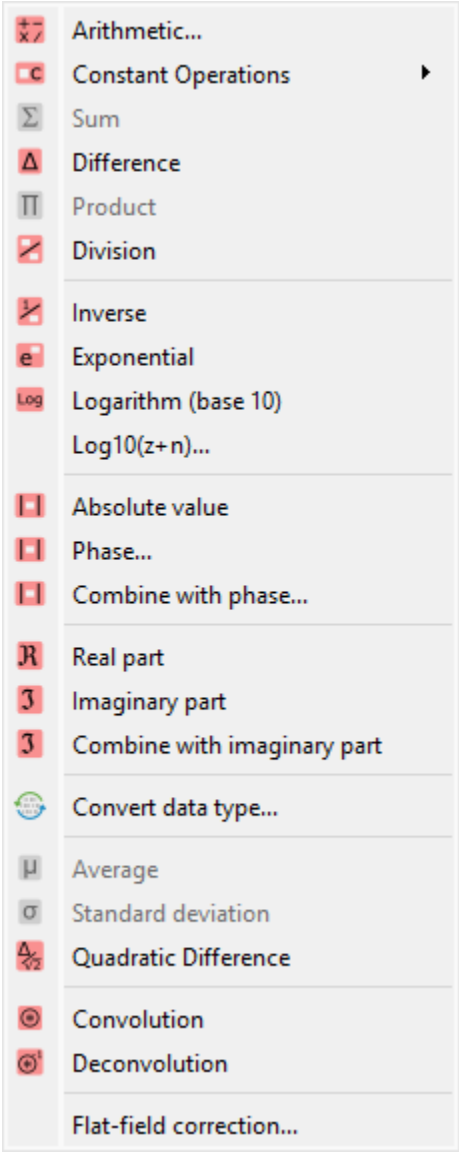


Fig. 52: Screenshot of the "Operations" menu.

Operations with a constant

Create a new image which is the result of a constant operation on each selected image:

Operation	Equation
Add	$z_k = z_{k-1} + conv(c)$
Subtract	$z_k = z_{k-1} - conv(c)$
Multiply	$z_k = conv(z_{k-1} \times c)$
Divide	$z_k = conv(\frac{z_{k-1}}{c})$

where c is the constant value and $conv$ is the conversion function which handles data type conversion (keeping the same data type as the input image).

Basic arithmetic operations

Arithmetic operations are performed pixel by pixel between the selected images.

Operation	Description
Sum	$z_M = \sum_{k=0}^{M-1} z_k$
Difference	$z_2 = z_1 - z_0$
Product	$z_M = \prod_{k=0}^{M-1} z_k$
Division	$z_2 = \frac{z_1}{z_0}$
Inverse	$z_2 = \frac{1}{z_1}$
Exponential	$z_2 = \exp(z_1)$
Logarithm (base 10)	$z_2 = \log_{10}(z_1)$

Basic mathematical functions

Function	Description
Exponential	$z_k = \exp(z_k)$
Logarithm (base 10)	$z_k = \log_{10}(z_k)$
Log10(z+10)	$z_k = \log_{10}(z_k + n)$ (useful if image background is zero)

Convolution and Deconvolution

Operation	Implementation
Convolution	Based on scipy.signal.convolve
Deconvolution	Frequency domain deconvolution

Absolute value and complex image operations

Operation	Description
Absolute value	$z_k = \lvert z_{k-1} \rvert$
Phase (argument)	`sigima.proc.image.phase()`
Combine with phase	Consider current image as the module and allow to select a image representing the phase to merge them in a complex image `sigima.proc.image.complex_from_magnitude_phase()`
Real part	$z_k = \Re(z_{k-1})$
Imaginary part	$z_k = \Im(z_{k-1})$
Combine with imaginary part	Consider current image as the real part and allow to select a image representing the imaginary part to merge them in a complex image `sigima.proc.image.complex_from_real_imag()`

Data type conversion

The "Convert data type" action allows you to convert the data type of the selected images. For integer data types, the conversion is done by clipping the values to the new data type range before effectively converting the data type. For floating point data types, the conversion is straightforward.

Note: Data type conversion uses the `sigima.tools.datatypes.clip_astype()` function which relies on `numpy.ndarray.astype()` function with the default parameters (*casting='unsafe'*).

Statistics between images

Create a new image which is the result of a statistical operation on each pixel of the selected images:

Operation	Description
Average	$z_M = \frac{1}{M} \sum_{k=0}^{M-1} z_k$
Standard Deviation	$z_M = \sqrt{\frac{1}{M} \sum_{k=0}^{M-1} (y_k - \bar{y})^2}$
Quadratic difference	$z_2 = \frac{z_1 - z_0}{\sqrt{2}}$

Flat-field correction

Create a new image which is flat-field correction of the **two** selected images:

$$z_1 = \begin{cases} \frac{z_0}{z_f}.\overline{z_f} & \text{if } z_0 > z_{threshold} \\ z_0 & \text{otherwise} \end{cases},$$

where z_0 is the raw image, z_f is the flat field image, $z_{threshold}$ is an adjustable threshold and $\overline{z_f}$ is the flat field image average value:

$$\overline{z_f} = \frac{1}{N_{row}.N_{col}}.\sum_{i=0}^{N_{row}} \sum_{j=0}^{N_{col}} z_f(i,j)$$

Note: Raw image and flat field image are supposedly already corrected by performing a dark frame subtraction.

2.3.6 Processing Images

This section describes the image processing features available in DataLab.

See also:

Operations on Images for more information on operations that can be performed on images, or *Analysis features on Images* for information on analysis features on images.

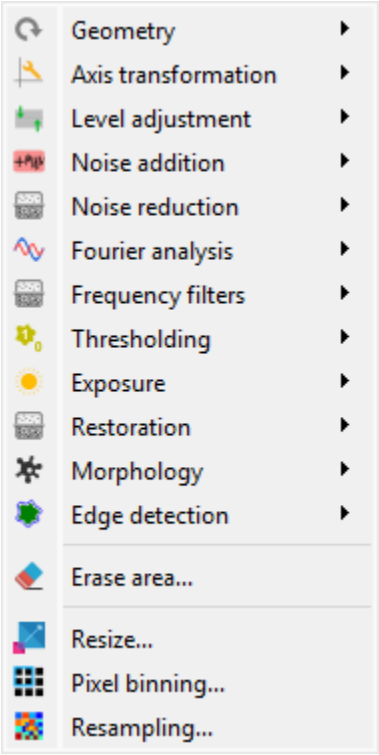


Fig. 53: Screenshot of the "Processing" menu.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "Processing" menu allows you to perform various processing on the current image or group of images: it allows you to apply filters, to perform exposure correction, to perform denoising, to perform morphological operations, and so on.

Geometry

Flip and rotation

Create a new image by flipping or rotating the data of the selected image. The image may be flipped horizontally, vertically, or diagonally (transposition). It may be rotated by 90°, 270° or any user-defined value.

Operation	Description
Flip horizontally	Mirror image along the vertical axis
Flip diagonally	Transpose the image (swap X/Y axes)
Flip vertically	Mirror image along the horizontal axis
Rotate 90° right	Rotate image 90° clockwise
Rotate 90° left	Rotate image 90° counter-clockwise
Rotate by…	Rotate image by any angle (user-defined)

Distribute images along a grid

Feature	Description
Distribute on a grid	Distribute selected images on a regular grid
Reset image positions	Reset the positions of the selected images to first image (x0, y0) coordinates

Axis transformation

Set uniform coordinates

Create a new image with uniform coordinates (i.e., with a constant pixel size in both X and Y directions).

Polynomial calibration

Create a new image which is a polynomial calibration of each selected image with respect to Z axis.

Note: Linear calibration was removed from the UI as it is a special case of polynomial calibration with degree 1.

Level adjustment

Normalize

Create a new image which is the normalized version of each selected image by maximum, amplitude, sum, energy or RMS:

Normalization	Equation
Maximum	$z_1 = \frac{z_0}{z_{\max}}$
Amplitude	$z_1 = \frac{z_0}{z_{\max} - z_{\min}}$
Area	$z_1 = \frac{z_0}{\sum_{i=0}^{N-1} z_i}$
Energy	$z_1 = \frac{z_0}{\sqrt{\sum_{n=0}^{N} \lvert z_0[n] \rvert^2}}$
RMS	$z_1 = \frac{z_0}{\sqrt{\frac{1}{N} \sum_{n=0}^{N} \lvert z_0[n] \rvert^2}}$

Clipping

Apply the clipping to each selected image.

Offset correction

Create a new image which is the result of offset correction on each selected image. This operation is performed by subtracting the image background value which is estimated by the mean value of a user-defined rectangular area.

Noise addition

Generate new images by adding the same noise to each selected image. The available noise types are:

Noise	Description
Gaussian	Normal distribution
Uniform	Uniform distribution
Poisson	Poisson distribution

Noise reduction

Create a new image which is the result of noise reduction on each selected image.

The following filters are available:

Filter	Formula/implementation
Gaussian filter	scipy.ndimage.gaussian_filter
Moving average	scipy.ndimage.uniform_filter
Moving median	scipy.ndimage.median_filter
Wiener filter	scipy.signal.wiener

Fourier analysis

Zero padding

Create a new image which is the result of zero padding on each selected image.

The following parameters are available:

Parameter	Description
Strategy	Zero padding strategy (see below)
Rows	Number of rows to add (if *strategy* is 'custom')
Columns	Number of columns to add (if *strategy* is 'custom')
Position	Position of the added zeros: 'bottom-right', 'centered'

Zero padding strategy refers to the method used to add zeros to the image, and it can be one of the following:

Strategy	Description
next_pow2	Next power of 2 (e.g. 512, 1024, ...)
multiple_of_64	Next multiple of 64 (e.g. 512, 576, ...)
custom	Custom size (user-defined)

FFT related functions

Create a new image which is the result of a Fourier analysis on each selected image.

The following functions are available:

Function	Description	Formula/implementation
FFT	Fast Fourier Transform	numpy.fft.fft2
Inverse FFT	Inverse Fast Fourier Transform	numpy.fft.ifft2
Magnitude spectrum	Optional: output in decibels (dB)	$z_1 = \lvert\mathrm{FFT}\left(z_0\right)\rvert$ or $z_1 = 20\log_{10}\left(\lvert\mathrm{FFT}\left(z_0\right)\rvert\right)$ (dB)
Phase spectrum	Phase of the FFT expressed in degrees, using numpy.angle	$z_1 = \angle\left(\mathrm{FFT}\left(z_0\right)\right)$
Power spectral density	Optional: output in decibels (dB)	$z_1 = \lvert\mathrm{FFT}\left(z_0\right)\rvert^2$ or $z_1 = 10\log_{10}\left(\lvert\mathrm{FFT}\left(z_0\right)\rvert^2\right)$ (dB)

Note: FFT and inverse FFT are performed using frequency shifting if the option is enabled in DataLab settings (see *Settings*).

Frequency-domain filters

The following frequency-domain filters are available:

Method	Description
Butterworth	Butterworth filter, based on skimage.filters.butterworth
Gaussian filter	Gaussian filter

Thresholding

Create a new image which is the result of thresholding on each selected image, eventually based on user-defined parameters ("Parametric thresholding").

The following parameters are available when selecting "Parametric thresholding":

Parameter	Description
Threshold method	The thresholding method to use (see table below)
Bins	Number of bins for histogram calculation
Value	Threshold value
Operation	Operation to apply (> or <)

The following thresholding methods are available:

Method	Implementation
Manual	Manual thresholding (user-defined parameters)
ISODATA	skimage.filters.threshold_isodata
Li	skimage.filters.threshold_li
Mean	skimage.filters.threshold_mean
Minimum	skimage.filters.threshold_minimum
Otsu	skimage.filters.threshold_otsu
Triangle	skimage.filters.threshold_triangle
Yen	skimage.filters.threshold_yen

Note: The "All thresholding methods" option allows to perform all thresholding methods on the same image. Combined with the "distribute on a grid" option, this allows to compare the different thresholding methods on the same image.

Exposure

Create a new image which is the result of exposure correction on each selected image.

The following functions are available:

Function	Implementation	Comments
Gamma correction	skimage.exposure.adjust_gamma	
Logarithmic correction	skimage.exposure.adjust_log	
Sigmoid correction	skimage.exposure.adjust_sigmoi	
Histogram equalization	skimage.exposure.equalize_hist	
Adaptive histogram equalization	skimage.exposure.equalize_adap	Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm
Intensity rescaling	skimage.exposure.rescale_intens	Stretch or shrink image intensity levels

Restoration

Create a new image which is the result of restoration on each selected image.

The following functions are available:

Function	Implementation	Comments
Total variation denoising	skimage.restoration.denoise_tv_	
Bilateral filter denoising	skimage.restoration.denoise_bila	
Wavelet denoising	skimage.restoration.denoise_wav	
White Top-Hat denoising	skimage.morphology.white_toph	Denoise image by subtracting its white top hat transform

Note: The "All denoising methods" option allows to perform all denoising methods on the same image. Combined with the "distribute on a grid" option, this allows to compare the different denoising methods on the same image.

Morphology

Create a new image which is the result of morphological operations on each selected image, using a disk footprint.

The following functions are available:

Function	Implementation
White Top-Hat (disk)	skimage.morphology.white_tophat
Black Top-Hat (disk)	skimage.morphology.black_tophat
Erosion (disk)	skimage.morphology.erosion
Dilation (disk)	skimage.morphology.dilation
Opening (disk)	skimage.morphology.opening
Closing (disk)	skimage.morphology.closing

Note: The "All morphological operations" option allows to perform all morphological operations on the same image. Combined with the "distribute on a grid" option, this allows to compare the different morphological operations on the same image.

Edges

Create a new image which is the result of edge filtering on each selected image.

The following functions are available:

Function	Implementation
Canny filter	skimage.feature.canny
Farid filter	skimage.filters.farid
Farid filter (horizontal)	skimage.filters.farid_h
Farid filter (vertical)	skimage.filters.farid_v
Laplace filter	skimage.filters.laplace
Prewitt filter	skimage.filters.prewitt
Prewitt filter (horizontal)	skimage.filters.prewitt_h
Prewitt filter (vertical)	skimage.filters.prewitt_v
Roberts filter	skimage.filters.roberts
Scharr filter	skimage.filters.scharr
Scharr filter (horizontal)	skimage.filters.scharr_h
Scharr filter (vertical)	skimage.filters.scharr_v
Sobel filter	skimage.filters.sobel
Sobel filter (horizontal)	skimage.filters.sobel_h
Sobel filter (vertical)	skimage.filters.sobel_v

Note: The "All edges filters" option allows to perform all edge filtering algorithms on the same image. Combined with the "distribute on a grid" option, this allows to compare the different edge filters on the same image.

Erase area

Erase an area in the image as defined by a region of interest (ROI).

Note: The region to erase is defined by the user through a dialog box. It can consist of a single ROI or multiple ROIs with various shapes (rectangular, circular, etc.). Note that the ROI defined here is not bound to any object; it is used solely to specify the area to erase in the image. In particular, it is independent of the image's ROI, if any (the latter is shown in the dialog box as a masked area).

Resize

Create a new image which is a resized version of each selected image.

Pixel binning

Combine clusters of adjacent pixels, throughout the image, into single pixels. The result can be the sum, average, median, minimum, or maximum value of the cluster.

Resampling

Generate new images by resampling each selected image. The following parameters are available:

Parameter	Description
math:*x_{min}*	Minimum x-coordinate of the output image
x_{max}	Maximum x-coordinate of the output image
y_{min}	Minimum y-coordinate of the output image
y_{max}	Maximum y-coordinate of the output image
Mode	Image size definition mode: 'Pixel size' or 'Output shape'. The 'Pixel size' mode allows to define the pixel size of the new image, while the 'Output shape' mode allows to define the number of pixels of the new image.
X	Pixel size in x-direction (if 'Pixel size' mode is selected)
Y	Pixel size in y-direction (if 'Pixel size' mode is selected)
Width	Output image width in pixels (if 'Output shape' mode is selected)
Height	Output image height in pixels (if 'Output shape' mode is selected)
Interpolation method	Interpolation method to use: 'nearest', 'linear', 'cubic'
Fill value	Value to use for points outside the input image domain (if None, function uses NaN for extrapolation)

2.3.7 Analysis features on Images

This section describes the image analysis features available in DataLab.

See also:

Operations on Images for more information on operations that can be performed on images, or *Processing Images* for information on processing features on images.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "Analysis" menu allows you to perform various computations on the current image or group of images. It also allows you to compute statistics, to compute the centroid, to detect peaks, to detect contours, and so on.

Note: In DataLab vocabulary, an "analysis" is a feature that computes a scalar result from an image. This result is stored as metadata, and thus attached to image. This is different from a "processing" which creates a new image from an existing one.

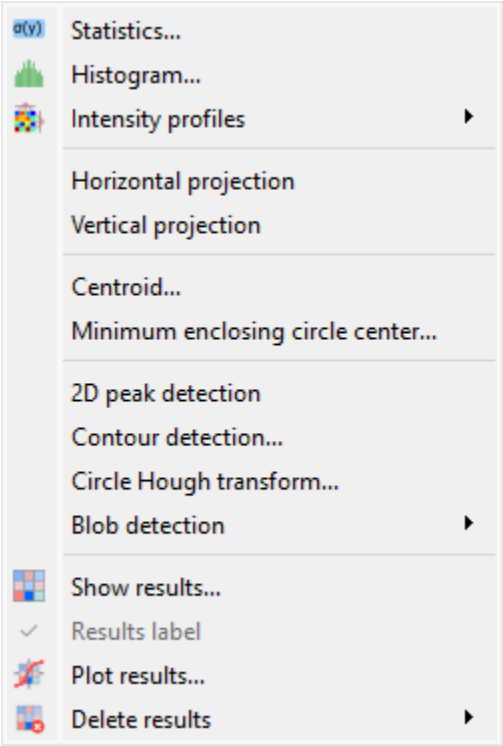


Fig. 54: Screenshot of the "Analysis" menu.

Statistics

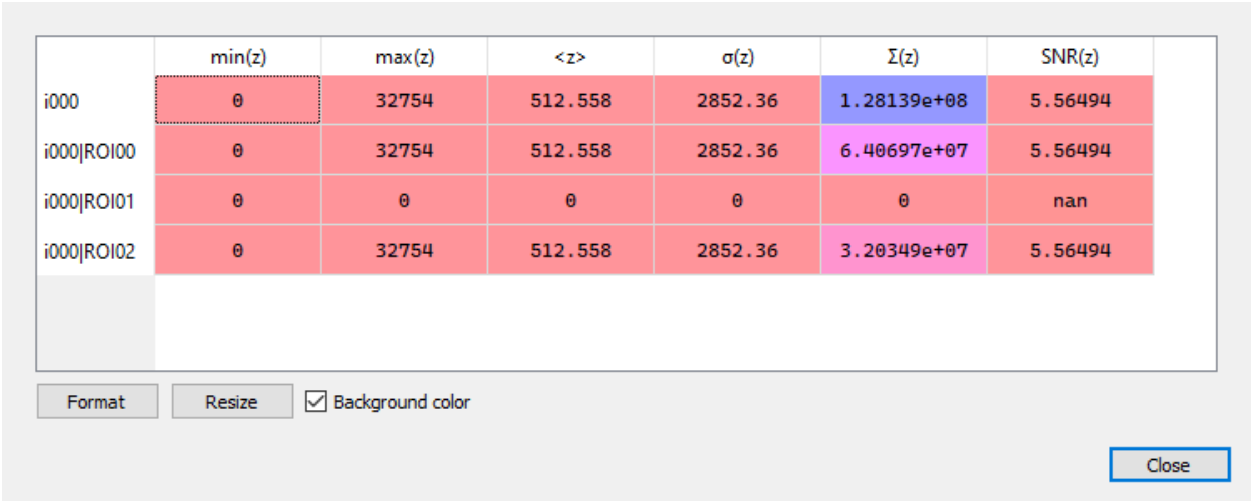
Compute statistics on selected image and show a summary table.

Histogram

Compute histogram of selected image and show it in the Signal Panel.

Parameters are:

Parameter	Description
Bins	Number of bins
Lower limit	Lower limit of the histogram
Upper limit	Upper limit of the histogram



	min(z)	max(z)	<z>	σ(z)	Σ(z)	SNR(z)
i000	0	32754	512.558	2852.36	1.28139e+08	5.56494
i000|ROI00	0	32754	512.558	2852.36	6.40697e+07	5.56494
i000|ROI01	0	0	0	0	0	nan
i000|ROI02	0	32754	512.558	2852.36	3.20349e+07	5.56494

Fig. 55: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

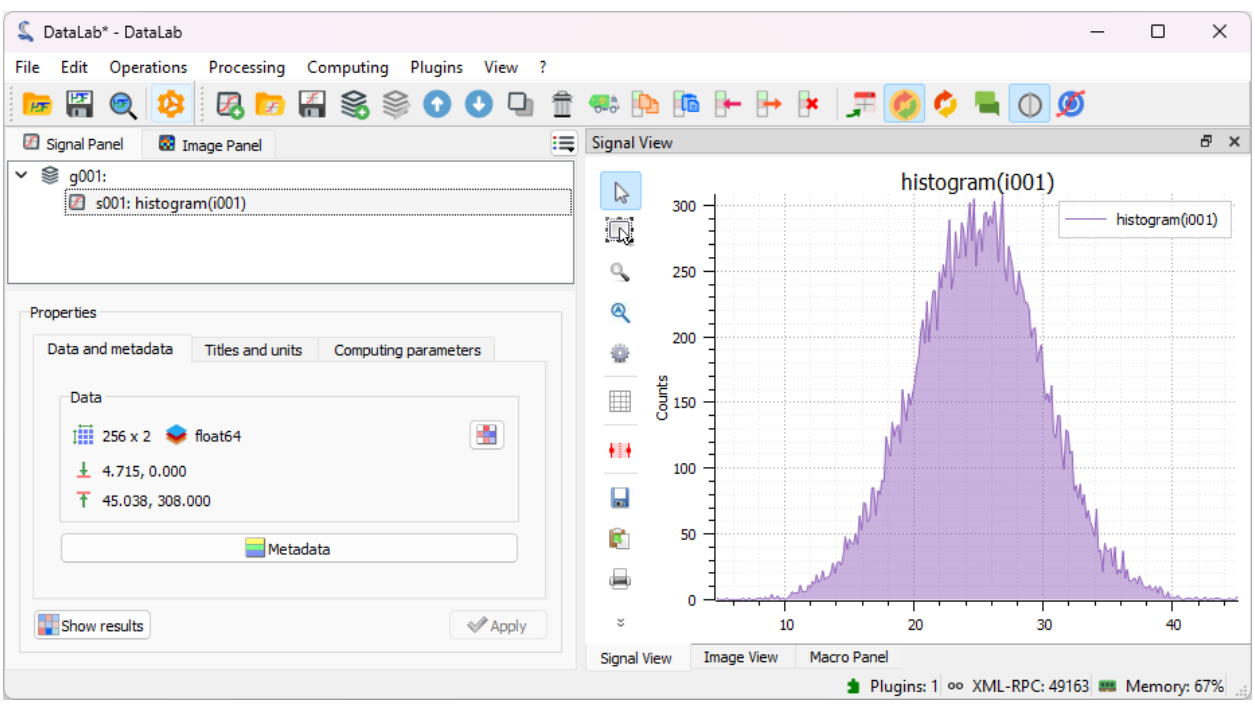


Fig. 56: Example of histogram.

Intensity profiles

Line profile

Extract an horizontal or vertical profile from each selected image, and create new signals from these profiles.

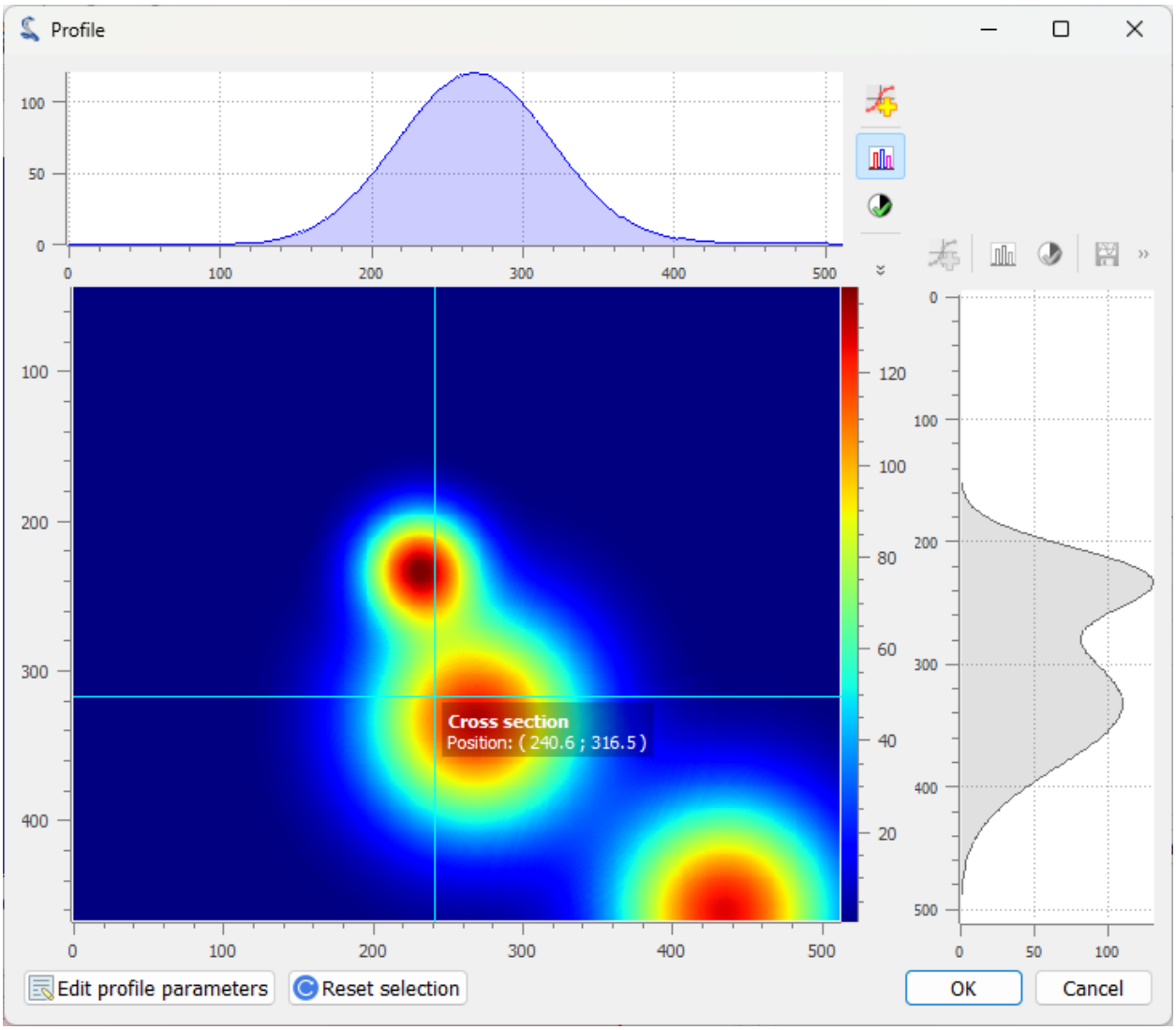


Fig. 57: Line profile dialog. Parameters may also be set manually ("Edit profile parameters" button).

Segment profile

Extract a segment profile from each selected image, and create new signals from these profiles.

Average profile

Extract an horizontal or vertical profile averaged over a rectangular area, from each selected image, and create new signals from these profiles.

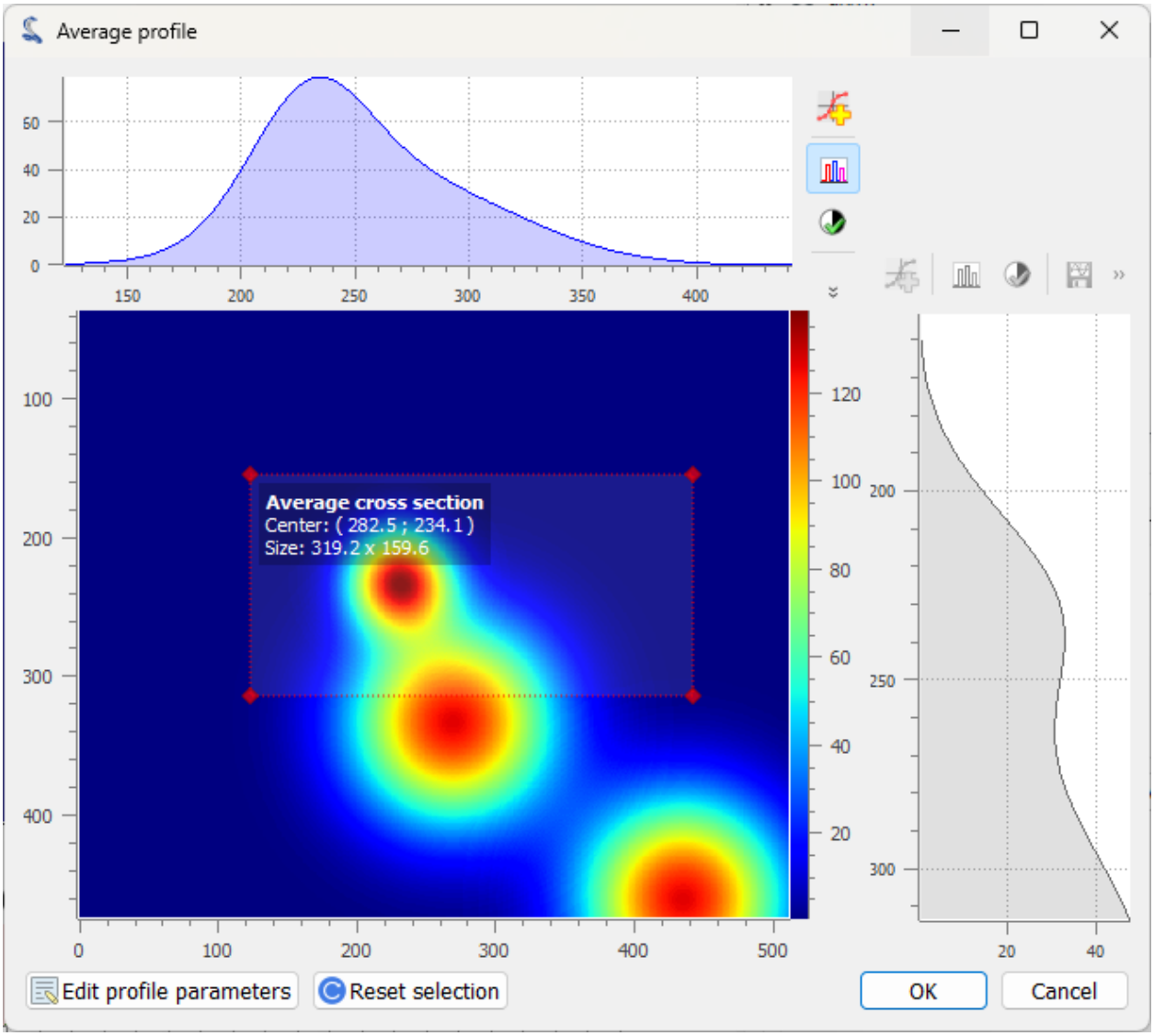


Fig. 58: Average profile dialog: the area is defined by a rectangle shape. Parameters may also be set manually ("Edit profile parameters" button).

Radial profile extraction

Extract a radial profile from each selected image, and create new signals from these profiles.

The following parameters are available:

Parameter	Description
Center	Center around which the radial profile is computed: centroid, image center, or user-defined
X	X coordinate of the center (if user-defined), in pixels
Y	Y coordinate of the center (if user-defined), in pixels

Horizontal and vertical projections

Compute horizontal and vertical projection profiles:

- Horizontal projection: Sum the pixel values along each row (projection on the x-axis).
- Vertical projection: Sum the pixel values along each column (projection on the y-axis).

Centroid

Compute the centroid of the image using a robust adaptive algorithm.

DataLab uses the `sigima.tools.image.get_centroid_auto()` function to estimate the image centroid.

This function combines the robustness of a Fourier-based approach (as discussed by Weisshaar et al.) with fallback mechanisms designed to handle pathological cases, such as:

- truncated or asymmetric shapes,
- off-center objects,
- strong background noise.

Internally, `sigima.tools.image.get_centroid_auto()` compares the Fourier result with two alternative estimators (a projected profile-based method `sigima.tools.image.get_projected_profile_centroid()` and a standard centroid from *scikit-image*), and selects the most consistent one.

This strategy ensures accurate and stable results across a wide range of image types — from clean laboratory data to noisy or partial acquisitions.

Minimum enclosing circle center

Compute the circle contour enclosing image values above a threshold level defined as the half-maximum value.

2D peak detection

Automatically find peaks on image using a minimum-maximum filter algorithm.

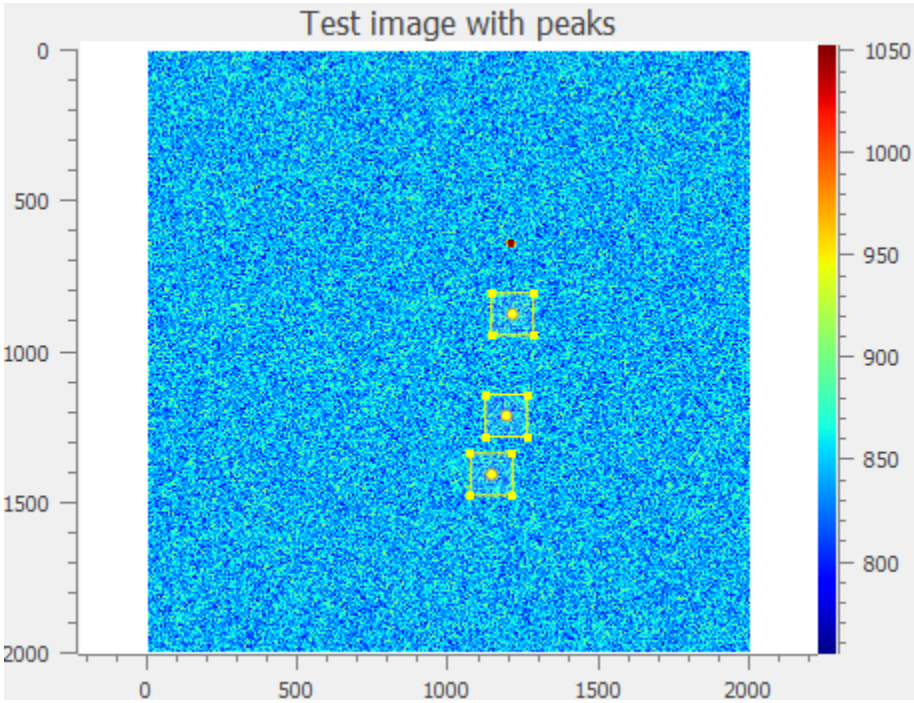


Fig. 59: Example of 2D peak detection.

See also:

See *2D Peak Detection* for more details on algorithm and associated parameters.

Contour detection

Automatically extract contours and fit them using a circle or an ellipse, or directly represent them as a polygon.

See also:

See *Contour Detection* for more details on algorithm and associated parameters.

Note: Computed scalar results are systematically stored as metadata. Metadata is attached to image and serialized with it when exporting current session in a HDF5 file.

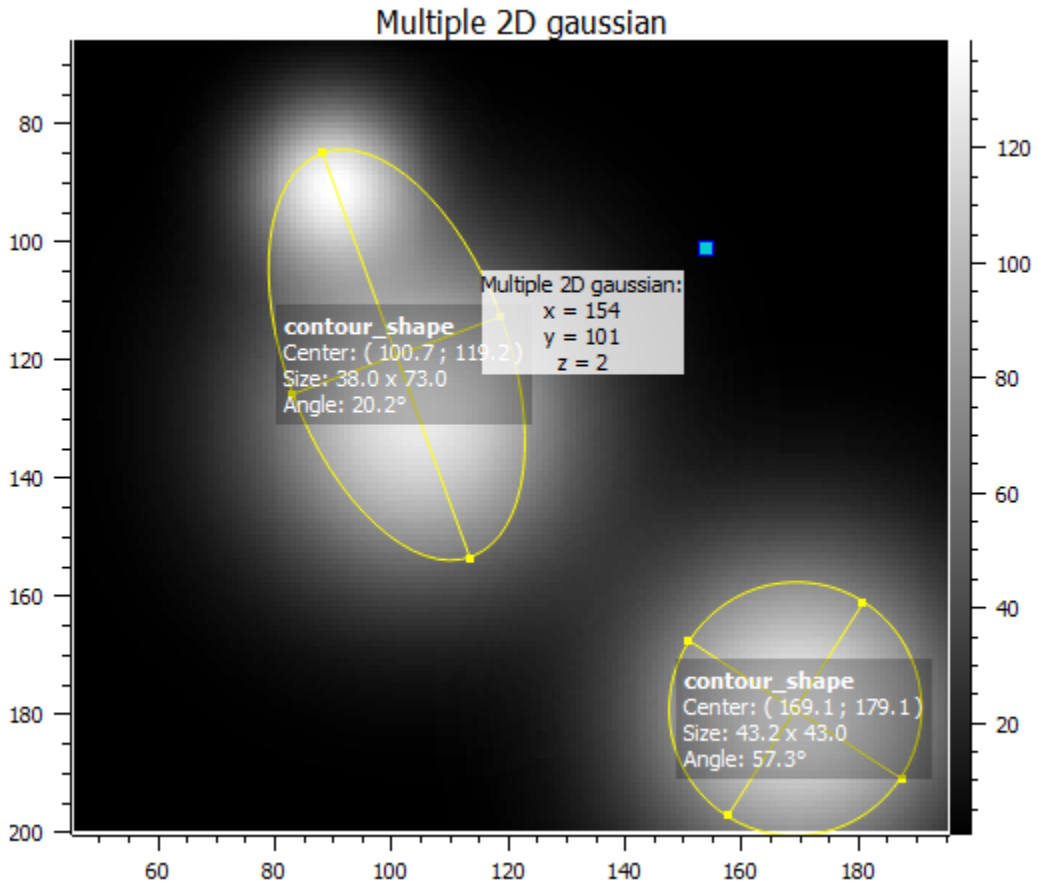


Fig. 60: Example of contour detection.

Circle Hough transform

Detect circular shapes using circle Hough transform (implementation based on skimage.transform.hough_circle_peaks).

Blob detection

Blob detection (DOG)
Detect blobs using Difference of Gaussian (DOG) method (implementation based on skimage.feature.blob_dog).

Blob detection (DOH)
Detect blobs using Determinant of Hessian (DOH) method (implementation based on skimage.feature.blob_doh).

Blob detection (LOG)
Detect blobs using Laplacian of Gaussian (LOG) method (implementation based on skimage.feature.blob_log).

Blob detection (OpenCV)
Detect blobs using OpenCV implementation of SimpleBlobDetector.

Note: Automatic ROI creation for detection features

All detection features (2D peak detection, contour detection, circle Hough transform, and blob detection) support automatic ROI creation around detected objects. This feature:

- Creates rectangular or circular ROIs around each detected feature
- Automatically sizes ROIs based on minimum distance between detections
- Requires at least 2 detected objects to determine appropriate ROI size
- Enables subsequent processing on individual detected regions

To use this feature, enable "Create regions of interest" in the detection parameters dialog and choose the desired ROI geometry.

Show results

Show the results of all analyses performed on the selected images. This shows the same table as the one shown after having performed a computation.

Results label

Toggle the visibility of result labels on the plot. When enabled, this checkable menu item displays result annotations (such as centroid markers, detected contours, blob circles, or other analysis shapes) directly on the image plot.

This option is synchronized between Signal and Image panels and persists across sessions. It is only enabled when results are available for the selected image.

Plot results

Plot the results of analyses performed on the selected images, with user-defined X and Y axes (e.g. plot the contour circle radius as a function of the image number).

2.3.8 View options for Images

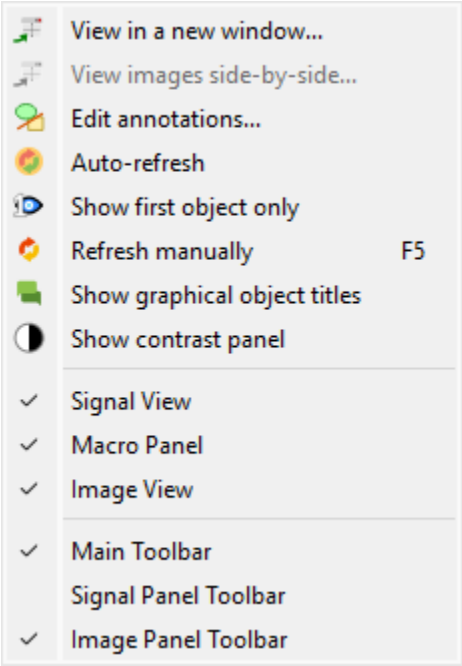


Fig. 61: Screenshot of the "View" menu.

When the "Image Panel" is selected, the menus and toolbars are updated to provide image-related actions.

The "View" menu allows you to visualize the current image or group of images. It also allows you to show/hide titles, to show/hide the contrast panel, to refresh the visualization, and so on.

View in a new window

Open a new window to visualize the selected images.

This option allows you to show the selected images in a separate window, in which you can visualize the data more comfortably (e.g., by maximizing the window) as well as to add or edit annotations.

When you click on the button "Annotations" in the toolbar of the new window, the annotation editing mode is activated (see the section "Edit annotations" below).

View images side-by-side

Open a new window to visualize the selected images side-by-side.

This option allows you to show the selected images in a separate window, arranged in a grid layout, with synchronized zooming and panning. This is useful to compare multiple images simultaneously.

Edit annotations

Open a new window to edit annotations.

This option allows you to edit the annotations of the selected images in a separate window. This is equivalent to select the "View in a new window" option and then click on the "Annotations" button in the toolbar of the new window.

Annotations are used to add text, lines, rectangles, ellipses, and other geometrical shapes to the images. They are useful to highlight regions of interest, to add comments, to mark points, and so on.

Note: The annotations are saved in the metadata of the image, so they are persistent and will be displayed every time you visualize the image.

The typical workflow to edit annotations is as follows:

1. Select the "Edit annotations" option.
2. In the new window, add annotations by clicking on the corresponding buttons at the bottom of the window.
3. Eventually, customize the annotations by changing their properties (e.g., the text, the color, the position, etc.) using the "Parameters" option in the context menu of the annotations.
4. When you are done, click on the "OK" button to save the annotations. This will close the window and the annotations will be saved in the metadata of the image and will be displayed in the main window.

Note: Annotations may be copied from an image to another by using the "copy/paste metadata" features.

Auto-refresh

Automatically refresh the visualization when the data changes:

- When enabled (default), the plot view is automatically refreshed when the data changes.
- When disabled, the plot view is not refreshed until you manually refresh it by using the "Refresh manually" menu entry.

Even though the refresh algorithm is optimized, it may still take some time to refresh the plot view when the data changes, especially when the data set is large. Therefore, you may want to disable the auto-refresh feature when you are working with large data sets, and enable it again when you are done. This will avoid unnecessary refreshes.

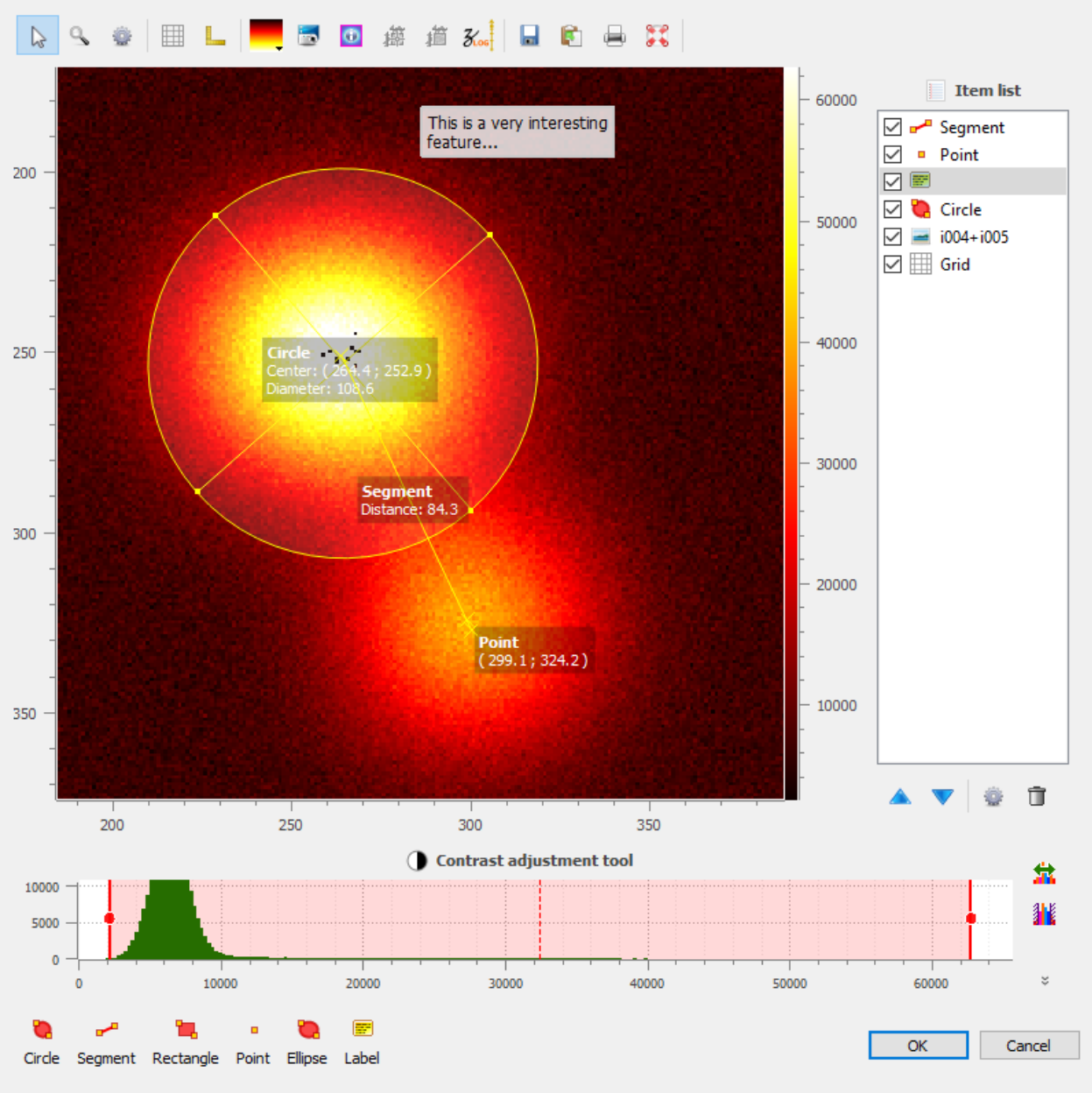


Fig. 62: Annotations may be added in the separate view.

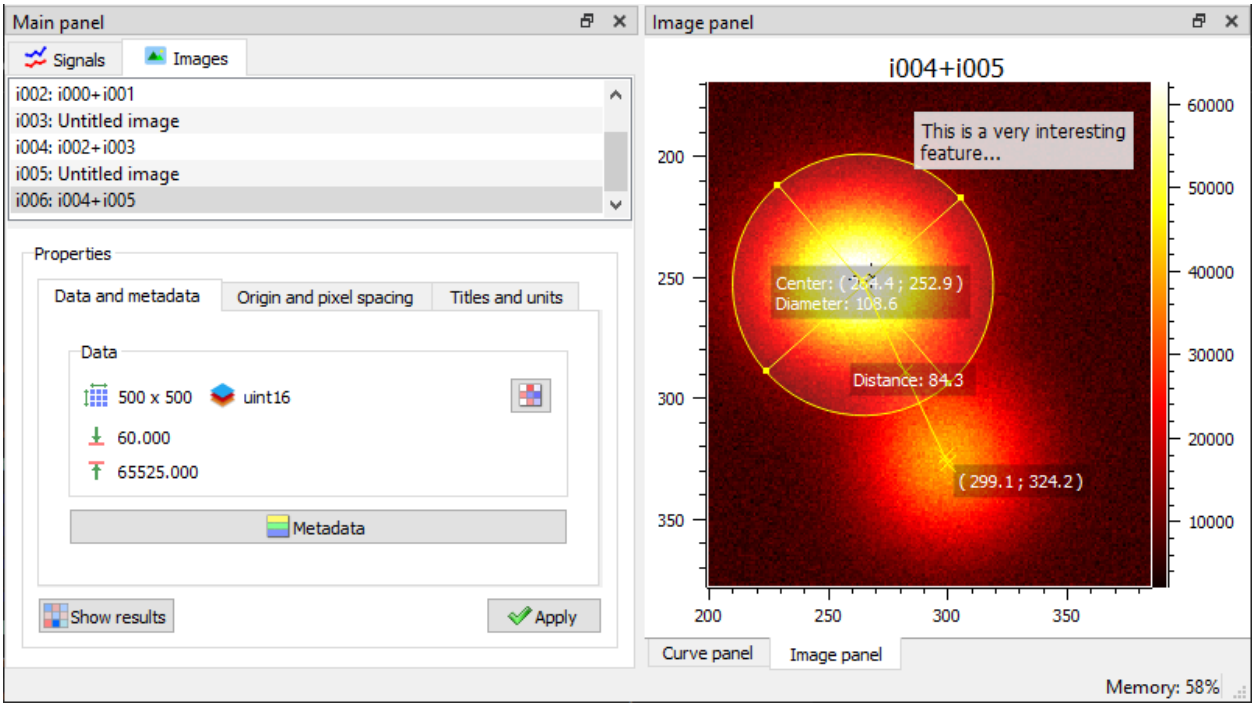


Fig. 63: Annotations are now part of the image metadata.

Show only first selected image

Toggle between showing all selected images or only the first one.

When this option is enabled, only the first selected image is displayed in the plot view. This may be useful when multiple images are selected and focusing on a single image is (temporarily) preferred.

Refresh manually

Refresh the visualization manually.

This triggers a refresh of the plot view. It is useful when the auto-refresh feature is disabled, or when you want to force a refresh of the plot view.

Show contrast panel

Show/hide contrast adjustment panel.

Show graphical object titles

Show/hide titles of analysis results or annotations.

This option allows you to show or hide the titles of the graphical objects (e.g., the titles of the analysis results or annotations). Hiding the titles can be useful when you want to visualize the data without any distractions, or if there are too many titles and they are overlapping.

2.3.9 2D Peak Detection

DataLab provides a "2D Peak Detection" feature which is based on a minimum-maximum filter algorithm.

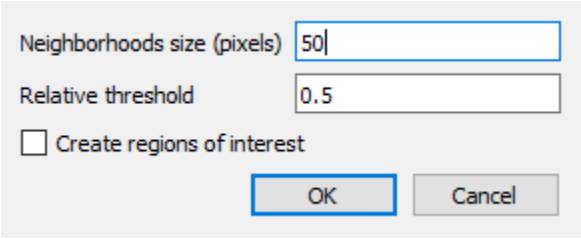


Fig. 64: 2D peak detection parameters.

How to use the feature:

- Create or open an image in DataLab workspace
- Select "2d peak detection" in "Analysis" menu
- Enter parameters "Neighborhoods size" and "Relative threhold"
- Optionally, enable "Create regions of interest" to automatically create ROIs around each detected peak:
  - Choose ROI geometry: "Rectangle" or "Circle"
  - ROI size is automatically calculated based on the minimum distance between detected peaks (to avoid overlap)
  - This feature requires at least 2 detected peaks
  - Created ROIs can be useful for subsequent processing on each peak area (e.g., contour detection, measurements, etc.)

Results are shown in a table:

- Each row is associated to a detected peak
- First column shows the ROI index (0 if no ROI is defined on input image)
- Second and third columns show peak coordinates

The 2d peak detection algorithm works in the following way:

- First, the minimum and maximum filtered images are computed using a sliding window algorithm with a user-defined size (implementation based on scipy.ndimage.minimum_filter and scipy.ndimage.maximum_filter)

Fig. 65: 2d peak detection results (see test "peak2d_app.py")

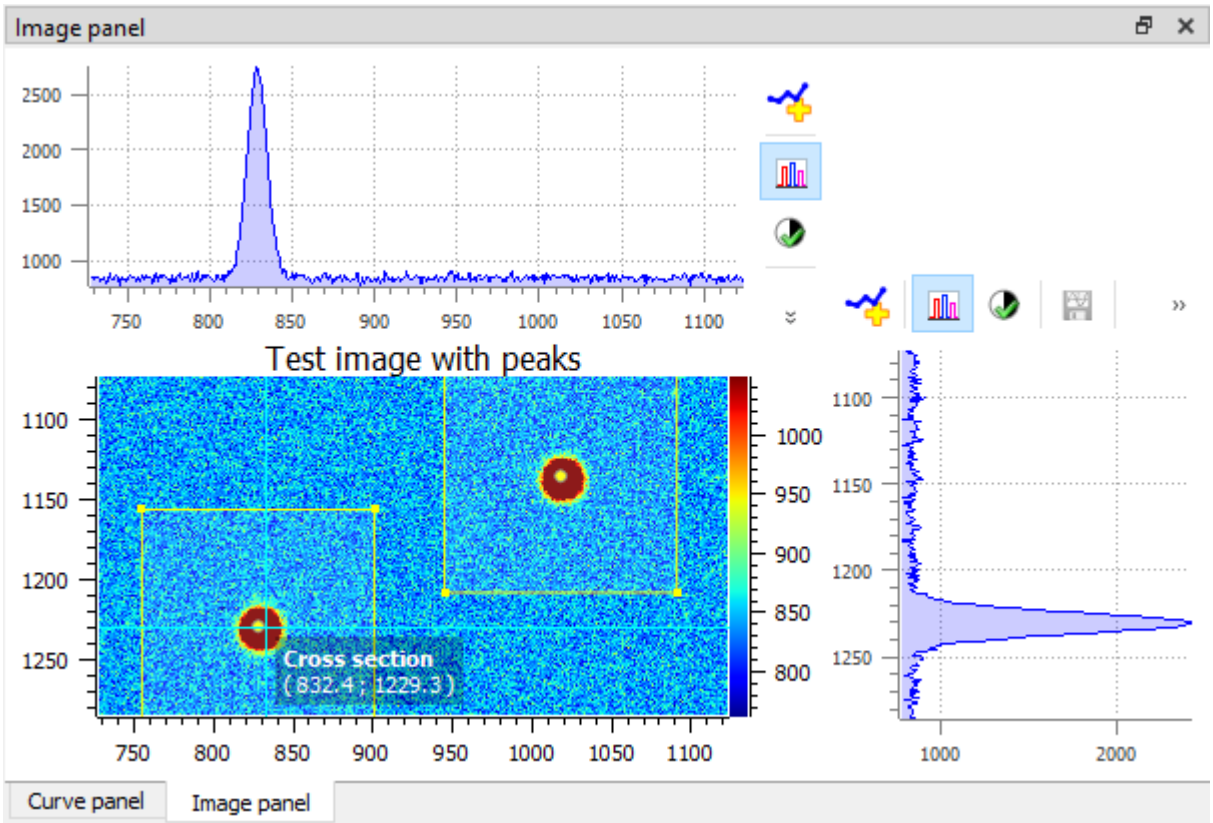


Fig. 66: Example of 2D peak detection.

- Then, the difference between the maximum and minimum filtered images is clipped at a user-defined threshold
- Resulting image features are labeled using scipy.ndimage.label
- Peak coordinates are then obtained from labels center
- Duplicates are eventually removed

The 2d peak detection parameters are the following:

- "Neighborhoods size": size of the sliding window (see above)
- "Relative threshold": detection threshold
- "Create regions of interest": if enabled, automatically creates ROIs around each detected peak (requires at least 2 peaks)
- "ROI geometry": shape of the ROIs ("Rectangle" or "Circle")

Feature is based on `get_2d_peaks_coords` function from `sigima.tools` module:

```
@check_2d_array(non_constant=True)
def get_2d_peaks_coords(
    data: np.ndarray, size: int | None = None, level: float = 0.5
) -> np.ndarray:
    """Detect peaks in image data, return coordinates.

    If neighborhoods size is None, default value is the highest value
    between 50 pixels and the 1/40th of the smallest image dimension.

    Detection threshold level is relative to difference
    between data maximum and minimum values.

    Args:
        data: Input data
        size: Neighborhood size (default: None)
        level: Relative level (default: 0.5)

    Returns:
        Coordinates of peaks
    """
    if size is None:
        size = max(min(data.shape) // 40, 50)
    data_max = spi.maximum_filter(data, size)
    data_min = spi.minimum_filter(data, size)
    data_diff = data_max - data_min
    diff = (data_max - data_min) > get_absolute_level(data_diff, level)
    maxima = data == data_max
    maxima[diff == 0] = 0
    labeled, _num_objects = spi.label(maxima)
    slices = spi.find_objects(labeled)
    coords = []
    for dy, dx in slices:
        x_center = int(0.5 * (dx.start + dx.stop - 1))
        y_center = int(0.5 * (dy.start + dy.stop - 1))
        coords.append((x_center, y_center))
    if len(coords) > 1:
```

(continues on next page)

(continued from previous page)

```
        # Eventually removing duplicates
        dist = distance_matrix(coords)
        for index in reversed(np.unique(np.where((dist < size) & (dist >␣
↪0))[1])):
            coords.pop(index)
    return np.array(coords)
```

2.3.10 Contour Detection

DataLab provides a "Contour Detection" feature which is based on the marching cubes algorithm.

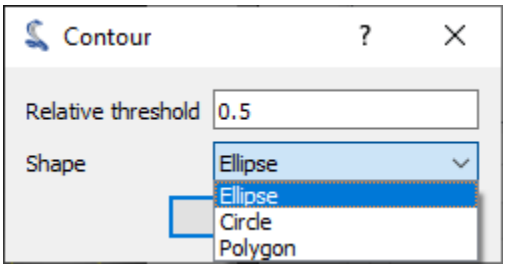


Fig. 67: Contour detection parameters.

How to use the feature:

- Create or open an image in DataLab workspace
- Eventually create a ROI around the target area
- Select "Contour detection" in "Analysis" menu
- Enter parameter "Shape" ("Ellipse", "Circle" or "Polygon")
- Optionally, enable "Create regions of interest" to automatically create ROIs around each detected contour:
  - Choose ROI geometry: "Rectangle" or "Circle"
  - ROI size is automatically calculated based on the minimum distance between detected contours (to avoid overlap)
  - This feature requires at least 2 detected contours
  - Created ROIs can be useful for subsequent processing on each contour area

Results are shown in a table:

- Each row is associated to a contour
- First column shows the ROI index (0 if no ROI is defined on input image)
- Other columns show contour coordinates: 4 columns for circles (coordinates of diameter), 8 columns for ellipses (coordinates of diameters)

The contour detection algorithm works in the following way:

- First, iso-valued contours are computed (implementation based on skimage.measure.find_contours.find_contours)
- Then, each contour is fitted to the closest ellipse (or circle)

Feature is based on `get_contour_shapes` function from `sigima.tools` module:

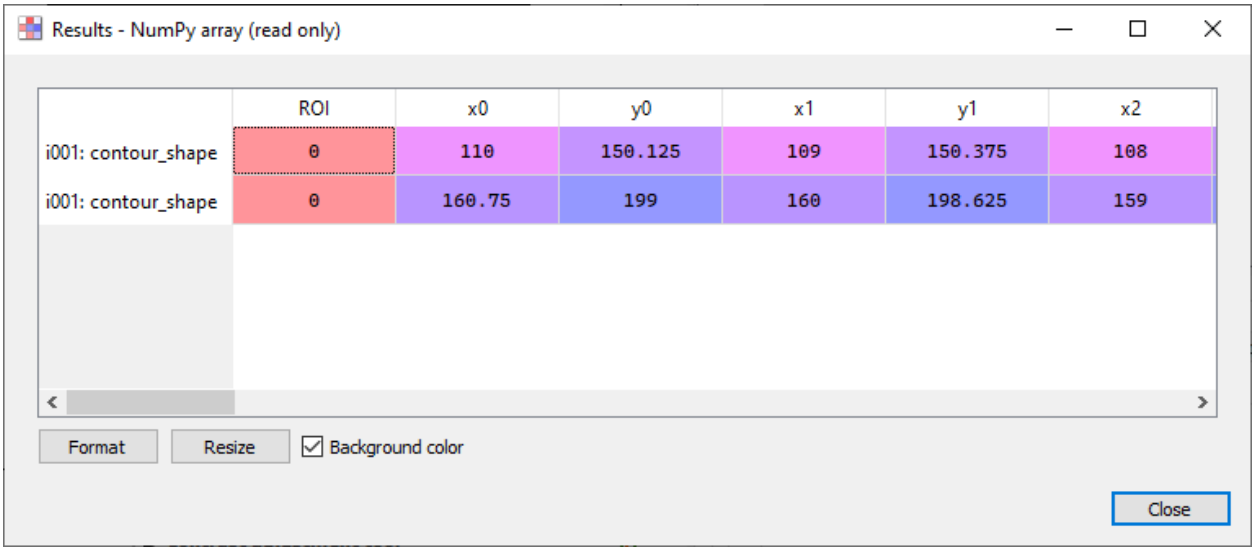


Fig. 68: Contour detection results (see test "contour_app.py")

```
def get_contour_shapes(
    data: np.ndarray | ma.MaskedArray,
    shape: ContourShape = ContourShape.ELLIPSE,
    level: float = 0.5,
) -> np.ndarray:
    """Find iso-valued contours in a 2D array, above relative level (.5 means␣
↪FWHM).

    Args:
        data: Input data
        shape: Shape to fit. Default is ELLIPSE
        level: Relative level (default: 0.5)

    Returns:
        Coordinates of shapes fitted to contours
    """
    # pylint: disable=too-many-locals
    contours = measure.find_contours(data, level=get_absolute_level(data,␣
↪level))
    coords = []
    for contour in contours:
        # `contour` is a (N, 2) array (rows, cols): we need to check if all␣
↪those
        # coordinates are masked: if so, we skip this contour
        if isinstance(data, ma.MaskedArray) and np.all(
            data.mask[contour[:, 0].astype(int), contour[:, 1].astype(int)]
        ):
            continue
        if shape == ContourShape.CIRCLE:
            model = measure.CircleModel()
            if model.estimate(contour):
                yc, xc, r = model.params
```

(continues on next page)

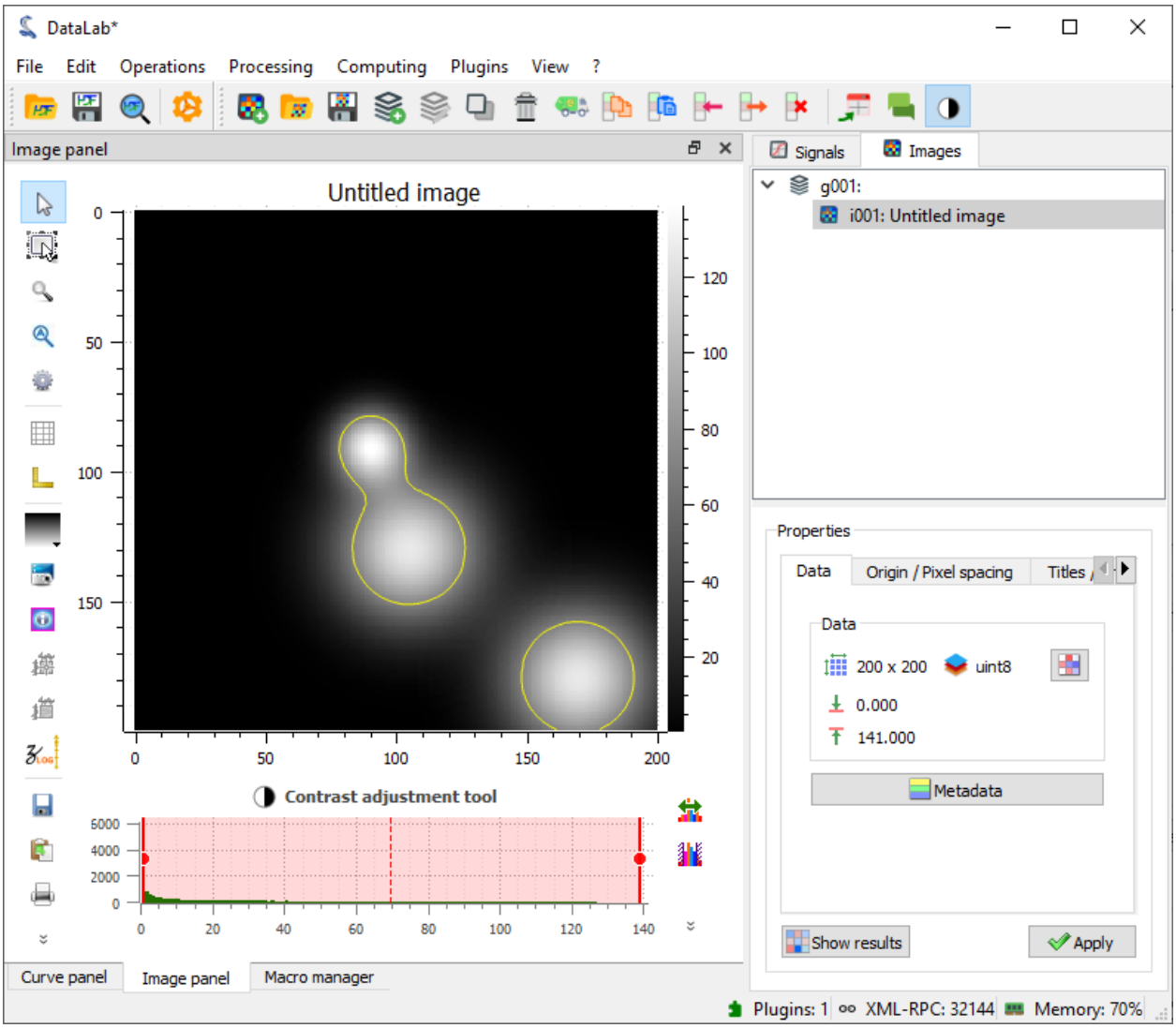


Fig. 69: Example of contour detection.

(continued from previous page)

```
                if r <= 1.0:
                    continue
                coords.append([xc, yc, r])
        elif shape == ContourShape.ELLIPSE:
            model = measure.EllipseModel()
            if model.estimate(contour):
                yc, xc, b, a, theta = model.params
                if a <= 1.0 or b <= 1.0:
                    continue
                coords.append([xc, yc, a, b, theta])
        elif shape == ContourShape.POLYGON:
            # `contour` is a (N, 2) array (rows, cols): we need to convert it
            # to a list of x, y coordinates flattened in a single list
            coords.append(contour[:, ::-1].flatten())
        else:
            raise NotImplementedError(f"Invalid contour model {model}")
    if shape == ContourShape.POLYGON:
        # `coords` is a list of arrays of shape (N, 2) where N is the number of␣
↪points
        # that can vary from one array to another, so we need to padd with␣
↪NaNs each
        # array to get a regular array:
        max_len = max(coord.shape[0] for coord in coords)
        arr = np.full((len(coords), max_len), np.nan)
        for i_row, coord in enumerate(coords):
            arr[i_row, : coord.shape[0]] = coord
        return arr
    return np.array(coords)
```

2.4 Validation

DataLab is a platform for scientific data analysis and visualization. It may be used in a variety of scientific disciplines, including biology, physics, and astronomy. The common ground for all these disciplines is the need to validate the results of computational analysis against ground-truth data. This is a critical step in the scientific method, and it is essential for reproducibility and trust in the results. This is what we call **technical validation**.

DataLab is also used in industrial applications, where the validation of the results is essential for guaranteeing the quality of the process but a wider range of validation is required to ensure that a maximum of use cases are covered from a functional point of view. This is what we call **functional validation**.

Thus, DataLab validation may be categorized into two types:

- **Technical validation**: ensures that the results of computational analysis are accurate and reliable.
- **Functional validation**: checks that the software behaves as expected in a variety of use cases (classic automated unit test suite).

2.4.1 Technical Validation

DataLab technical validation is based on two key concepts: **ground-truth data** and **analytical validation**.

Ground-truth data

Ground-truth data is data that is known to be correct. It is used to validate the results of computational analysis.

In DataLab, ground-truth data may be obtained from a variety of sources, including:

- Experimental data
- Simulated data
- Synthetic data
- Data from a trusted source

Analytical validation

Analytical validation is the process of comparing the results of computational analysis to ground-truth data. This is done to ensure that the results are accurate and reliable.

In DataLab, analytical validation is implemented using a variety of techniques, including:

- Cross-validation with an analytical model (from a trusted source, e.g. SciPy or NumPy)
- Statistical analysis
- Visual inspection
- Expert review

Scope

The scope of technical validation in DataLab includes all compute functions that operate on DataLab's signal and image objects (i.e. `sigima.objects.SignalObj` and `sigima.objects.ImageObj`).

This includes functions for (all functions are named `compute_<function_name>`):

- Signal processing (`sigima.proc.signal`)
- Image processing (`sigima.proc.image`)

Implementation

The tests are implemented using the pytest framework.

When writing a new technical validation test, the following rules should be followed regarding the test function:

- The test function should be named:
  - `test_signal_<function_name>` for signal compute functions
  - `test_image_<function_name>` for image compute functions

Note: The `signal` or `image` prefix is used to indicate the type of object that the function operates on. It may be omitted if the function operates exclusively on one type of object (e.g. `test_adjust_gamma` is the test function for the `compute_adjust_gamma` function, which operates on images).

- The test function should be marked with the `@pytest.mark.validation` decorator.

Following those rules ensures that:

- The tests are easily identified as technical validation tests.
- The tests can be executed separately using the command line interface (see *Run technical validation tests*).
- The tests are automatically discovered for synthetizing the validation status of the compute functions (see *Validation Status of DataLab*).

Executing tests

In DataLab, technical validation tests are disseminated in the test suite of the project, but they can also be executed separately using the command line interface.

See also:

See paragraph *Run technical validation tests* for more information on how to run technical validation tests.

2.4.2 Functional validation

Strategy

DataLab functional validation is based a classic test strategy, with a strong emphasis on automated testing. Apart from one or two manual tests (e.g. load test), all tests are automated (more than 99% of the tests are automated).

Writing tests follows the TDD (Test-Driven Development) principle:

- When *a new feature is developed*, the developer writes the tests first. The tests are then executed to ensure that they fail. The developer then implements the feature, and the tests are executed again to ensure that they pass.
- When *a bug is reported*, the developer writes a test that reproduces the bug. The test is executed to ensure that it fails. The developer then fixes the bug, and the test is executed again to ensure that it passes.

Depending on the abstraction level, unit tests and/or application tests are written. When writing both types of tests, the developer starts with the unit tests and then writes the application tests.

Types of tests

The functional validation of DataLab is based on two main types of tests:

- **Unit tests** (test scripts named `*_unit_test.py`): Test individual functions or methods. All unit tests are automated.
- **Application tests** (test scripts named `*_app_test.py`): Test the interaction between components (integration tests), or the application as a whole. All application tests are automated.

Implementation

The tests are implemented using the pytest framework. Many existing tests may be derived from to create new tests.

Executing tests

To execute the tests, the developer uses the command line interface. See section *Run complete test suite* for more information on how to run functional validation tests.

2.4.3 Validation Status of DataLab

Functional validation

In DataLab, functional validation is based on a classic test strategy (see *Functional validation*).

Test coverage is around 90%, with more than 200 tests.

Technical validation

This paragraph provides the validation status of compute functions in DataLab (this is what we call technical validation, see *Technical Validation*).

Note: This is a work in progress: the tables below are updated continuously as new functions are validated or test code is adapted (the tables are generated from the test code). Some functions are already validated but do not appear in the list below yet, while others are still in the validation process.

Warning: The validation status must not be confused with the test coverage. The validation status indicates whether the function has been validated against ground-truth data or analytical models. The test coverage indicates the percentage of the code that is executed by the test suite, but it does not necessarily take into account the correctness of the results (DataLab's test coverage is around 90%).

Table 3: Validation Statistics

Category	Signal	Image	Total
Number of compute functions	95	115	210
Number of validated compute functions	95	115	210
Percentage of validated compute functions	100%	100%	100%

Signal Compute Functions

The table below shows the validation status of signal compute functions in DataLab. It is automatically generated from the source code.

Table 4: Validation status of signal compute functions

Compute function	Description	Test function
absolute	Compute absolute value with numpy.absolute	test_signal_absolute
add_gaussian_noise	Add normal noise to the input signal	test_signal_add_gaussian_noise
add_poisson_noise	Add Poisson noise to the input signal	test_signal_add_poisson_noise
add_uniform_noise	Add uniform noise to the input signal	test_signal_add_uniform_noise
addition	Compute the element-wise sum of multiple signals	test_signal_addition
addition_constant	Compute the sum of a signal and a constant value	test_signal_addition_constant
allan_deviation	Compute Allan deviation	test_signal_allan_deviation
allan_variance	Compute Allan variance	test_signal_allan_variance
apply_window	Compute windowing	test_signal_apply_window
arithmetic	Perform an arithmetic operation on two signals	test_signal_arithmetic
astype	Convert data type	test_signal_astype
average	Compute the element-wise average of multiple signals	test_signal_average
bandpass	Compute band-pass filter	test_signal_bandpass
bandstop	Compute band-stop filter	test_signal_bandstop
bandwidth_3db	Compute bandwidth at -3 dB	test_signal_bandwidth_3db
calibration	Compute linear calibration	test_signal_calibration
cdf_fit	Compute CDF fit	test_signal_cdf_fit
clip	Compute maximum data clipping	test_signal_clip
complex_from_magnitude_pha	Combine magnitude and phase signals into a complex signal	test_signal_complex_from_magnitude_phase
complex_from_real_imag	Combine two real signals into a complex signal using real + i * imag	test_signal_complex_from_real_imag
contrast	Compute contrast	test_signal_contrast
convolution	Compute convolution of two signals	test_signal_convolution
deconvolution	Compute deconvolution	test_signal_deconvolution
derivative	Compute derivative	test_signal_derivative
detrending	Detrend data	test_signal_detrending
difference	Compute the element-wise difference between two signals	test_signal_difference
difference_constant	Compute the difference between a signal and a constant value	test_signal_difference_constant
division	Compute the element-wise division between two signals	test_signal_division
division_constant	Compute the division of a signal by a constant value	test_signal_division_constant
dynamic_parameters	Compute Dynamic parameters	test_dynamic_parameters
evaluate_fit	Evaluate fit function from src1 on the x-axis of src2	test_signal_evaluate_fit
exp	Compute exponential with numpy.exp	test_signal_exp
exponential_fit	Compute exponential fit	test_signal_exponential_fit

continues on next page

Table 4 – continued from previous page

Compute function	Description	Test function
extract_pulse_features	Extract pulse features	test_signal_extract_pulse_features
extract_roi	Extract single region of interest from data	test_signal_extract_roi
extract_rois	Extract multiple regions of interest from data	test_signal_extract_rois
fft	Compute FFT	test_signal_fft
full_width_at_y		test_signal_full_width_at_y
fw1e2	Compute FW at $1/e^2$	test_signal_fw1e2
fwhm	Compute FWHM	test_signal_fwhm
gaussian_filter	Compute gaussian filter	test_signal_gaussian_filter
gaussian_fit	Compute Gaussian fit	test_signal_gaussian_fit
hadamard_variance	Compute Hadamard variance	test_signal_hadamard_variance
highpass	Compute high-pass filter	test_signal_highpass
histogram	Compute histogram	test_signal_histogram
ifft	Compute the inverse FFT	test_signal_ifft
imag	Compute imaginary part	test_signal_imag
integral	Compute integral	test_signal_integral
interpolate	Interpolate data	test_signal_interpolate
inverse	Compute the element-wise inverse of a signal	test_signal_inverse
linear_fit	Compute linear fit	test_signal_linear_fit
log10	Compute Log10 with numpy.log10	test_signal_log10
lorentzian_fit	Compute Lorentzian fit	test_signal_lorentzian_fit
lowpass	Compute low-pass filter	test_signal_lowpass
magnitude_spectrum	Compute magnitude spectrum	test_signal_magnitude_spectrum
modified_allan_variance	Compute Modified Allan variance	test_signal_modified_allan_variance
moving_average	Compute moving average	test_signal_moving_average
moving_median	Compute moving median	test_signal_moving_median
normalize	Normalize data	test_signal_normalize
offset_correction	Correct offset: subtract the mean value of the signal in the specified range	test_signal_offset_correction
overlapping_allan_variance	Compute Overlapping Allan variance	test_signal_overlapping_allan_variance
peak_detection	Peak detection	test_signal_peak_detection
phase	Compute the phase (argument) of a complex signal	test_signal_phase
phase_spectrum	Compute phase spectrum	test_signal_phase_spectrum
piecewiseexponential_fit	Compute piecewise exponential fit (raise-decay)	test_signal_piecewiseexponential_fit
planckian_fit	Compute Planckian fit	test_signal_planckian_fit
polynomial_fit	Compute polynomial fit	test_polynomial_fit
power	Compute power with numpy.power	test_signal_power
product	Compute the element-wise product of multiple signals	test_signal_product
product_constant	Compute the product of a signal and a constant value	test_signal_product_constant
psd	Compute power spectral density	test_signal_psd
quadratic_difference	Compute the normalized difference between two signals	test_signal_quadratic_difference
real	Compute real part	test_signal_real
resampling	Resample data	test_signal_resampling
reverse_x	Reverse x-axis	test_signal_reverse_x
sampling_rate_period	Compute sampling rate and period	test_signal_sampling_rate_period

continues on next page

Table 4 – continued from previous page

Compute function	Description	Test function
sigmoid_fit	Compute sigmoid fit	test_signal_sigmoid_fit
signals_to_image	Combine multiple signals into an image	test_signal_signals_to_image
sinusoidal_fit	Compute sinusoidal fit	test_sinusoidal_fit
sqrt	Compute square root with numpy.sqrt	test_signal_sqrt
standard_deviation	Compute the element-wise standard deviation of multiple signals	test_signal_standard_deviation
stats	Compute statistics on a signal	test_signal_stats_unit
time_deviation	Compute Time Deviation (TDEV)	test_signal_time_deviation
to_cartesian	Convert polar coordinates to Cartesian coordinates	test_signal_to_cartesian
to_polar	Convert Cartesian coordinates to polar coordinates	test_signal_to_polar
total_variance	Compute Total variance	test_signal_total_variance
transpose	Transpose signal (swap X and Y axes)	test_signal_transpose
twohalfgaussian_fit	Compute two-half-Gaussian fit	test_signal_twohalfgaussian_fit
voigt_fit	Compute Voigt fit	test_signal_voigt_fit
wiener	Compute Wiener filter	test_signal_wiener
x_at_minmax		test_signal_x_at_minmax
x_at_y		test_signal_x_at_y
xy_mode	Simulate the X-Y mode of an oscilloscope	test_signal_xy_mode
y_at_x		test_signal_y_at_x
zero_padding	Compute zero padding	test_signal_zero_padding

Image Compute Functions

The table below shows the validation status of image compute functions in DataLab. It is automatically generated from the source code.

Table 5: Validation status of image compute functions

Compute function	Description	Test function
absolute	Compute absolute value with numpy.absolute	test_image_absolute
add_gaussian_noise	Add Gaussian (normal) noise to the input image	test_image_add_gaussian_noise
add_poisson_noise	Add Poisson noise to the input image	test_image_add_poisson_noise
add_uniform_noise	Add uniform noise to the input image	test_image_add_uniform_noise
addition	Add images in the list and return the result image object	test_image_addition
addition_constant	Add **dst** and a constant value and return the new result image object	test_image_addition_constant
adjust_gamma	Gamma correction	test_adjust_gamma
adjust_log	Compute log correction	test_adjust_log
adjust_sigmoid	Compute sigmoid correction	test_adjust_sigmoid
arithmetic	Compute arithmetic operation on two images	test_image_arithmetic
astype	Convert image data type	test_image_astype
average	Compute the average of images in the list and return the result image object	test_image_average

continues on next page

Table 5 – continued from previous page

Compute function	Description	Test function
average_profile	Compute horizontal or vertical average profile	test_average_profile
binning	Binning: image pixel binning (or aggregation)	test_binning
black_tophat	Compute Black Top-Hat	test_black_tophat
blob_dog	Compute blobs using Difference of Gaussian method	test_image_blob_dog
blob_doh	Compute blobs using Determinant of Hessian method	test_image_blob_doh
blob_log	Compute blobs using Laplacian of Gaussian method	test_image_blob_log
blob_opencv	Compute blobs using OpenCV	test_image_blob_opencv
butterworth	Compute Butterworth filter	test_butterworth
calibration	Compute polynomial calibration	test_image_calibration
canny	Compute Canny filter	test_canny
centroid	Compute centroid	test_image_centroid
clip	Apply clipping	test_image_clip
closing	Compute morphological closing	test_closing
complex_from_magnitude_pha	Combine magnitude and phase images into a complex image	test_image_complex_from_magnitude_phase
complex_from_real_imag	Combine two real images into a complex image using real + i * imag	test_image_complex_from_real_imag
contour_shape	Compute contour shape	test_contour_shape
convolution	Convolve an image with a kernel	test_image_convolution
deconvolution	Deconvolve a kernel from an image using Fast Fourier Transform (FFT)	test_image_deconvolution
denoise_bilateral	Compute bilateral filter denoising	test_denoise_bilateral
denoise_tophat	Denoise using White Top-Hat	test_denoise_tophat
denoise_tv	Compute Total Variation denoising	test_denoise_tv
denoise_wavelet	Compute Wavelet denoising	test_denoise_wavelet
difference	Compute difference between two images	test_image_difference
difference_constant	Subtract a constant value from an image and return the new result image object	test_image_difference_constant
dilation	Compute Dilation	test_dilation
division	Compute division between two images	test_image_division
division_constant	Divide an image by a constant value and return the new result image object	test_image_division_constant
enclosing_circle	Compute minimum enclosing circle	test_image_enclosing_circle
equalize_adapthist	Adaptive histogram equalization	test_equalize_adapthist
equalize_hist	Histogram equalization	test_equalize_hist
erase	Erase an area of the image using the mean value of the image	test_erase
erosion	Compute Erosion	test_erosion
exp	Compute exponential with numpy.exp	test_image_exp
extract_roi	Extract single ROI	test_image_extract_roi
extract_rois	Extract multiple regions of interest from data	test_image_extract_rois
farid	Compute Farid filter	test_farid
farid_h	Compute horizontal Farid filter	test_farid_h
farid_v	Compute vertical Farid filter	test_farid_v
fft	Compute FFT	test_image_fft

continues on next page

Table 5 – continued from previous page

Compute function	Description	Test function
flatfield	Compute flat field correction	test_flatfield
fliph	Flip data horizontally	test_image_fliph
flipv	Flip data vertically	test_image_flipv
gaussian_filter	Compute gaussian filter	test_image_gaussian_filter
gaussian_freq_filter	Apply a Gaussian filter in the frequency domain	test_gaussian_freq_filter
histogram	Compute histogram of the image data,	test_image_histogram
horizontal_projection	Compute the sum of pixel intensities along each col. (projection on the x-axis)	test_image_horizontal_projection
hough_circle_peaks	Compute Hough circles	test_image_hough_circle_peaks
ifft	Compute inverse FFT	test_image_ifft
imag	Compute imaginary part	test_image_imag
inverse	Compute the inverse of an image and return the new result image object	test_image_inverse
laplace	Compute Laplace filter	test_laplace
line_profile	Compute horizontal or vertical profile	test_line_profile
log10	Compute log10 with numpy.log10	test_image_log10
log10_z_plus_n	Compute log10(z+n) with numpy.log10	test_image_log10_z_plus_n
magnitude_spectrum	Compute magnitude spectrum	test_image_magnitude_spectrum
moving_average	Compute moving average	test_image_moving_average
moving_median	Compute moving median	test_image_moving_median
normalize		test_image_normalize
offset_correction	Apply offset correction	test_image_offset_correction
opening	Compute morphological opening	test_opening
peak_detection	Compute 2D peak detection	test_image_peak_detection
phase	Compute the phase (argument) of a complex image	test_image_phase
phase_spectrum	Compute phase spectrum	test_image_phase_spectrum
prewitt	Compute Prewitt filter	test_prewitt
prewitt_h	Compute horizontal Prewitt filter	test_prewitt_h
prewitt_v	Compute vertical Prewitt filter	test_prewitt_v
product	Multiply images in the list and return the result image object	test_image_product
product_constant	Multiply **dst** by a constant value and return the new result image object	test_image_product_constant
psd	Compute power spectral density	test_image_psd
quadratic_difference	Compute quadratic difference between two images	test_image_quadratic_difference
radial_profile	Compute radial profile around the centroid	test_radial_profile
real	Compute real part	test_image_real
resampling	Resample image to new coordinate grid using interpolation	test_image_resampling
rescale_intensity	Rescale image intensity levels	test_rescale_intensity
resize	Zooming function	test_image_resize
roberts	Compute Roberts filter	test_roberts
rotate	Rotate data	test_image_rotate
rotate270	Rotate data 270°	test_image_rotate270
rotate90	Rotate data 90°	test_image_rotate90
scharr	Compute Scharr filter	test_scharr
scharr_h	Compute horizontal Scharr filter	test_scharr_h

continues on next page

Table 5 – continued from previous page

Compute function	Description	Test function
scharr_v	Compute vertical Scharr filter	test_scharr_v
segment_profile	Compute segment profile	test_segment_profile
set_uniform_coords	Convert image to uniform coordinate system	test_set_uniform_coords
sobel	Compute Sobel filter	test_sobel
sobel_h	Compute horizontal Sobel filter	test_sobel_h
sobel_v	Compute vertical Sobel filter	test_sobel_v
standard_deviation	Compute the element-wise standard deviation of multiple images	test_image_standard_deviation
stats	Compute statistics on an image	test_image_stats_unit
threshold	Compute the threshold, using one of the available algorithms	test_threshold
threshold_isodata	Compute the threshold using the Isodata algorithm with default parameters	test_threshold_isodata
threshold_li	Compute the threshold using the Li algorithm with default parameters	test_threshold_li
threshold_mean	Compute the threshold using the Mean algorithm	test_threshold_mean
threshold_minimum	Compute the threshold using the Minimum algorithm with default parameters	test_threshold_minimum
threshold_otsu	Compute the threshold using the Otsu algorithm with default parameters	test_threshold_otsu
threshold_triangle	Compute the threshold using the Triangle algorithm with default parameters	test_threshold_triangle
threshold_yen	Compute the threshold using the Yen algorithm with default parameters	test_threshold_yen
translate	Translate data	test_image_translate
transpose	Transpose image	test_image_transpose
vertical_projection	Compute the sum of pixel intensities along each row (projection on the y-axis)	test_image_vertical_projection
white_tophat	Compute White Top-Hat	test_white_tophat
wiener	Compute Wiener filter	test_image_wiener
zero_padding	Zero-padding: add zeros to image borders	test_image_zero_padding

2.5 Advanced features

This section describes the advanced features of DataLab.

2.5.1 Plugins

DataLab supports a robust plugin architecture, allowing users to extend the application's features without modifying its core. Plugins can introduce new processing tools, data import/export formats, or custom GUI elements — all seamlessly integrated into the platform.

What is a plugin?

A plugin is a Python module that is automatically loaded by DataLab at startup. It can define new features or modify existing ones.

To be recognized as a plugin, the file must:

- Be a Python module whose name **starts with** `datalab_` (e.g. `datalab_myplugin.py`),
- Contain a class that **inherits from** *`datalab.plugins.PluginBase`*,
- Include a class attribute named `PLUGIN_INFO`, which must be an instance of *`datalab.plugins.PluginInfo`*.

This *PLUGIN_INFO* object is used by DataLab to retrieve metadata such as the plugin name, type, and menu integration.

Note: Only Python files whose names start with `datalab_` will be scanned for plugins.

DataLab supports three categories of plugins, each with its own purpose and registration mechanism:

- **Processing and visualization plugins** Add custom actions for signal or image processing. These may include new computation functions, data visualization tools, or interactive dialogs. Integrated into a dedicated submenu of the "Plugins" menu.
- **Input/Output plugins** Define new file formats (read and/or write) handled transparently by DataLab's I/O framework. These plugins extend compatibility with custom or third-party data formats.
- **HDF5 plugins** Special plugins that support HDF5 files with domain-specific tree structures. These allow DataLab to interpret signals or images organized in non-standard ways.

Where to put a plugin?

Plugins are automatically discovered at startup from multiple locations:

- The user plugin directory: Typically *~/.DataLab/plugins* on Linux/macOS or *C:/Users/YourName/.DataLab/plugins* on Windows.
- A custom plugin directory: Configurable in DataLab's preferences.
- The standalone distribution directory: If using a frozen (standalone) build, the *plugins* folder located next to the executable is scanned.
- The internal *datalab/plugins* folder (not recommended for user plugins): This location is reserved for built-in or bundled plugins and should not be modified manually.

How to develop a plugin?

The recommended approach to developing a plugin is to derive from an existing example and adapt it to your needs. You can explore the source code in the *datalab/plugins* folder or refer to community-contributed examples.

Note: Most of DataLab's signal and image processing functionalities have been externalized into a dedicated library called **Sigima** (https://sigima.readthedocs.io/en/latest/). When developing DataLab plugins, you will typically import and use many Sigima functions and features to perform data processing, analysis, and visualization tasks. Sigima provides a comprehensive set of tools for scientific data manipulation that can be leveraged directly in your plugins.

To develop in your usual Python environment (e.g., with an IDE like Spyder), you can:

1. **Install DataLab in your Python environment**, using one of the following methods:
   - *Conda package*
   - *Package manager pip*
   - *Wheel package*
   - *Source package*
2. **Or add the `datalab` package manually to your Python path**:
   - Download the source from the PyPI page,
   - Unzip the archive,
   - Add the *datalab* directory to your PYTHONPATH (e.g., using the *PYTHONPATH Manager* in Spyder).

Note: Even if you've installed *datalab* in your environment, you cannot run the full DataLab application directly from an IDE. You must launch DataLab via the command line or using the installer-created shortcut to properly test your plugin.

Example: processing plugin

Here is a minimal example of a plugin that prints a message when activated:

```
# Copyright (c) DataLab Platform Developers, BSD 3-Clause license, see LICENSE file.

"""
Test Data Plugin for DataLab
----------------------------

This plugin is an example of DataLab plugin. It provides test data samples
and some actions to test DataLab functionalities.
"""

from __future__ import annotations

import sigima.tests.data as test_data
from sigima.io.image import ImageIORegistry
from sigima.io.signal import SignalIORegistry
from sigima.tests import helpers
```

(continues on next page)

(continued from previous page)

```
from datalab.config import _
from datalab.plugins import PluginBase, PluginInfo
from datalab.utils.qthelpers import create_progress_bar

# ---------------------------------------------------------------------------
# All computation functions must be defined as global functions, otherwise
# they cannot be pickled and sent to the worker process
# ---------------------------------------------------------------------------


class PluginTestData(PluginBase):
    """DataLab Test Data Plugin"""

    PLUGIN_INFO = PluginInfo(
        name=_("Test data"),
        version="1.0.0",
        description=_("Testing DataLab functionalities"),
    )

    def load_test_objs(
        self, registry_class: type[SignalIORegistry | ImageIORegistry], title: str
    ) -> None:
        """Load all test objects from a given registry class

        Args:
            registry_class: Registry class (SignalIORegistry or ImageIORegistry)
            title: Progress bar title

        Returns:
            List of (filename, object) tuples
        """
        test_objs = list(helpers.read_test_objects(registry_class))
        with create_progress_bar(self.signalpanel, title, max_=len(test_objs)) as prog:
            for i_obj, (_fname, obj) in enumerate(test_objs):
                prog.setValue(i_obj + 1)
                if prog.wasCanceled():
                    break
                if obj is not None:
                    self.proxy.add_object(obj)

    # Signal processing features ------------------------------------------------
    def create_paracetamol_signal(self) -> None:
        """Create paracetamol signal"""
        obj = test_data.create_paracetamol_signal()
        self.proxy.add_object(obj)

    # Image processing features -------------------------------------------------
    def create_peak_image(self) -> None:
        """Create 2D peak image"""
        obj = self.imagepanel.new_object(add_to_panel=False)
        if obj is not None:
            param = test_data.PeakDataParam.create(size=max(obj.data.shape))
```

(continues on next page)

(continued from previous page)

```
        self.imagepanel.processor.update_param_defaults(param)
        if param.edit(self.main):
            obj.data, _coords = test_data.get_peak2d_data(param)
            self.proxy.add_object(obj)

    def create_sincos_image(self) -> None:
        """Create 2D sin cos image"""
        newparam = self.edit_new_image_parameters(hide_type=True)
        if newparam is not None:
            obj = test_data.create_sincos_image(newparam)
            self.proxy.add_object(obj)

    def create_noisy_gaussian_image(self) -> None:
        """Create 2D noisy gauss image"""
        newparam = self.edit_new_image_parameters(hide_height=True, hide_type=True)
        if newparam is not None:
            obj = test_data.create_noisy_gaussian_image(newparam, add_annotations=False)
            self.proxy.add_object(obj)

    def create_multigaussian_image(self) -> None:
        """Create 2D multi gauss image"""
        newparam = self.edit_new_image_parameters(hide_height=True, hide_type=True)
        if newparam is not None:
            obj = test_data.create_multigaussian_image(newparam)
            self.proxy.add_object(obj)

    def create_2dstep_image(self) -> None:
        """Create 2D step image"""
        newparam = self.edit_new_image_parameters(hide_type=True)
        if newparam is not None:
            obj = test_data.create_2dstep_image(newparam)
            self.proxy.add_object(obj)

    def create_ring_image(self) -> None:
        """Create 2D ring image"""
        param = test_data.RingParam(_("Ring"))
        if param.edit(self.main):
            obj = test_data.create_ring_image(param)
            self.proxy.add_object(obj)

    def create_annotated_image(self) -> None:
        """Create annotated image"""
        obj = test_data.create_annotated_image()
        self.proxy.add_object(obj)

    def create_grid_gaussian_image(self) -> None:
        """Create image with a grid of gaussian spots"""
        param = test_data.GridOfGaussianImages(_("Grid of Gaussian Images"))
        if param.edit(self.main):
            obj = test_data.create_grid_of_gaussian_images(param)
            self.proxy.add_object(obj)
```

(continues on next page)

(continued from previous page)

```
    # Plugin menu entries -------------------------------------------------------
    def create_actions(self) -> None:
        """Create actions"""
        # Signal Panel ---------------------------------------------------------
        sah = self.signalpanel.acthandler
        with sah.new_menu(_("Test data")):
            sah.new_action(
                _("Load spectrum of paracetamol"),
                triggered=self.create_paracetamol_signal,
                select_condition="always",
            )
            sah.new_action(
                _("Load all test signals"),
                triggered=lambda regclass=SignalIORegistry,
                title=_("Load all test signals"): self.load_test_objs(regclass, title),
                select_condition="always",
                separator=True,
            )
        # Image Panel ----------------------------------------------------------
        iah = self.imagepanel.acthandler
        with iah.new_menu(_("Test data")):
            iah.new_action(
                _("Create image with peaks"),
                triggered=self.create_peak_image,
                select_condition="always",
                separator=True,
            )
            iah.new_action(
                _("Create 2D sin cos image"),
                triggered=self.create_sincos_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D noisy gaussian image"),
                triggered=self.create_noisy_gaussian_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D multi gaussian image"),
                triggered=self.create_multigaussian_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create annotated image"),
                triggered=self.create_annotated_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D step image"),
                triggered=self.create_2dstep_image,
                select_condition="always",
            )
```

(continues on next page)

(continued from previous page)

```
            iah.new_action(
                _("Create ring image"),
                triggered=self.create_ring_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create image with a grid of gaussian spots"),
                triggered=self.create_grid_gaussian_image,
                select_condition="always",
            )
            iah.new_action(
                _("Load all test images"),
                triggered=lambda regclass=ImageIORegistry,
                title=_("Load all test images"): self.load_test_objs(regclass, title),
                select_condition="always",
                separator=True,
            )
```

Example: input/output plugin

Here is a simple example of a plugin that adds a new file formats to DataLab.

```
# Copyright (c) DataLab Platform Developers, BSD 3-Clause license, see LICENSE file.

"""
Image file formats Plugin for DataLab
-------------------------------------

This plugin is an example of DataLab plugin.
It provides image file formats from cameras, scanners, and other acquisition devices.
"""

import struct

import numpy as np
from sigima.io.base import FormatInfo
from sigima.io.image.base import SingleImageFormatBase

# ==============================================================================
# Thales Pixium FXD file format
# ==============================================================================


class FXDFile:
    """Class implementing Thales Pixium FXD Image file reading feature

    Args:
        fname (str): path to FXD file
        debug (bool): debug mode
    """
```

(continues on next page)

(continued from previous page)

```
    HEADER = "<lllllllffl"

    def __init__(self, fname: str = None, debug: bool = False) -> None:
        self.__debug = debug
        self.file_format = None  # long
        self.nbcols = None  # long
        self.nbrows = None  # long
        self.nbframes = None  # long
        self.pixeltype = None  # long
        self.quantlevels = None  # long
        self.maxlevel = None  # float
        self.minlevel = None  # float
        self.comment_length = None  # long
        self.fname = None
        self.data = None
        if fname is not None:
            self.load(fname)

    def __repr__(self) -> str:
        """Return a string representation of the object"""
        info = (
            ("Image width", f"{self.nbcols:d}"),
            ("Image Height", f"{self.nbrows:d}"),
            ("Frame number", f"{self.nbframes:d}"),
            ("File format", f"{self.file_format:d}"),
            ("Pixel type", f"{self.pixeltype:d}"),
            ("Quantlevels", f"{self.quantlevels:d}"),
            ("Min. level", f"{self.minlevel:f}"),
            ("Max. level", f"{self.maxlevel:f}"),
            ("Comment length", f"{self.comment_length:d}"),
        )
        desc_len = max(len(d) for d in list(zip(*info))[0]) + 3
        res = ""
        for description, value in info:
            res += ("{:" + str(desc_len) + "}{}\n").format(description + ": ", value)

        res = object.__repr__(self) + "\n" + res
        return res

    def load(self, fname: str) -> None:
        """Load header and image pixel data

        Args:
            fname (str): path to FXD file
        """
        with open(fname, "rb") as data_file:
            header_s = struct.Struct(self.HEADER)
            record = data_file.read(9 * 4)
            unpacked_rec = header_s.unpack(record)
            (
                self.file_format,
                self.nbcols,
```

(continues on next page)

(continued from previous page)

```
                self.nbrows,
                self.nbframes,
                self.pixeltype,
                self.quantlevels,
                self.maxlevel,
                self.minlevel,
                self.comment_length,
            ) = unpacked_rec
            if self.__debug:
                print(unpacked_rec)
                print(self)
            data_file.seek(128 + self.comment_length)
            if self.pixeltype == 0:
                size, dtype = 4, np.float32
            elif self.pixeltype == 1:
                size, dtype = 2, np.uint16
            elif self.pixeltype == 2:
                size, dtype = 1, np.uint8
            else:
                raise NotImplementedError(f"Unsupported pixel type: {self.pixeltype}")
            block = data_file.read(self.nbrows * self.nbcols * size)
        data = np.frombuffer(block, dtype=dtype)
        self.data = data.reshape(self.nbrows, self.nbcols)


class FXDImageFormat(SingleImageFormatBase):
    """Object representing Thales Pixium (FXD) image file type"""

    FORMAT_INFO = FormatInfo(
        name="Thales Pixium",
        extensions="*.fxd",
        readable=True,
        writeable=False,
    )

    @staticmethod
    def read_data(filename: str) -> np.ndarray:
        """Read data and return it

        Args:
            filename (str): path to FXD file

        Returns:
            np.ndarray: image data
        """
        fxd_file = FXDFile(filename)
        return fxd_file.data


# ==============================================================================
# Dürr NDT XYZ file format
# ==============================================================================
```

(continues on next page)

(continued from previous page)

```
class XYZImageFormat(SingleImageFormatBase):
    """Object representing Dürr NDT XYZ image file type"""

    FORMAT_INFO = FormatInfo(
        name="Dürr NDT",
        extensions="*.xyz",
        readable=True,
        writeable=True,
    )

    @staticmethod
    def read_data(filename: str) -> np.ndarray:
        """Read data and return it

        Args:
            filename (str): path to XYZ file

        Returns:
            np.ndarray: image data
        """
        with open(filename, "rb") as fdesc:
            cols = int(np.fromfile(fdesc, dtype=np.uint16, count=1)[0])
            rows = int(np.fromfile(fdesc, dtype=np.uint16, count=1)[0])
            arr = np.fromfile(fdesc, dtype=np.uint16, count=cols * rows)
            arr = arr.reshape((rows, cols))
        return np.fliplr(arr)

    @staticmethod
    def write_data(filename: str, data: np.ndarray) -> None:
        """Write data to file

        Args:
            filename: File name
            data: Image array data
        """
        data = np.fliplr(data)
        with open(filename, "wb") as fdesc:
            fdesc.write(np.array(data.shape[1], dtype=np.uint16).tobytes())
            fdesc.write(np.array(data.shape[0], dtype=np.uint16).tobytes())
            fdesc.write(data.tobytes())
```

Other examples

Other examples of plugins can be found in the *plugins/examples* directory of the DataLab source code (explore here on GitHub).

Migrating from v0.20 to v1.0

If you have existing plugins written for DataLab v0.20, please refer to the *migration guide* for detailed instructions on updating your plugins to work with DataLab v1.0.

Public API

DataLab plugin system

DataLab plugin system provides a way to extend the application with new functionalities.

Plugins are Python modules that relies on two classes:

- `PluginInfo`, which stores information about the plugin
- `PluginBase`, which is the base class for all plugins

Plugins may also extends DataLab I/O features by providing new image or signal formats. To do so, they must provide a subclass of `ImageFormatBase` or `SignalFormatBase`, in which format information is defined using the `FormatInfo` class.

class datalab.plugins.**PluginRegistry**(*name*, *bases*, *attrs*)

Metaclass for registering plugins

classmethod get_plugin_classes() → list[type[*PluginBase*]]

Return plugin classes

classmethod get_plugins() → list[*PluginBase*]

Return plugin instances

classmethod get_plugin(*name_or_class: str | type[*PluginBase*]*) → *PluginBase* | None

Return plugin instance

classmethod register_plugin(*plugin:* PluginBase)

Register plugin

classmethod unregister_plugin(*plugin:* PluginBase)

Unregister plugin

classmethod unregister_all_plugins()

Unregister all plugins

classmethod get_plugin_info(*html: bool = True*) → str

Return plugin information (names, versions, descriptions) in html format

Parameters

html – return html formatted text (default: True)

class datalab.plugins.**PluginInfo**(*name: str | None = None*, *version: str = '0.0.0'*, *description: str = ''*, *icon: str | None = None*)

Plugin info

class datalab.plugins.**PluginBaseMeta**(*name*, *bases*, *namespace*, ***kwargs*)

Mixed metaclass to avoid conflicts

class datalab.plugins.**PluginBase**

Plugin base class

property signalpanel: *SignalPanel*

Return signal panel

property imagepanel: *ImagePanel*

Return image panel

show_warning(*message: str*)

Show warning message

show_error(*message: str*)

Show error message

show_info(*message: str*)

Show info message

ask_yesno(*message: str*, *title: str | None = None*, *cancelable: bool = False*) → bool

Ask yes/no question

edit_new_signal_parameters(*title: str | None = None*, *size: int | None = None*) → NewSignalParam

Create and edit new signal parameter dataset

Parameters

- **title** – title of the new signal
- **size** – size of the new signal (default: None, get from current signal)

Returns

New signal parameter dataset (or None if canceled)

edit_new_image_parameters(*title: str | None = None*, *shape: tuple[int, int] | None = None*, *hide_height: bool = False*, *hide_width: bool = False*, *hide_type: bool = True*, *hide_dtype: bool = False*) → NewImageParam | None

Create and edit new image parameter dataset

Parameters

- **title** – title of the new image
- **shape** – shape of the new image (default: None, get from current image)
- **hide_height** – hide image heigth parameter (default: False)
- **hide_width** – hide image width parameter (default: False)
- **hide_type** – hide image type parameter (default: True)
- **hide_dtype** – hide image data type parameter (default: False)

Returns

New image parameter dataset (or None if canceled)

is_registered()

Return True if plugin is registered

register(*main:* main.DLMainWindow) → None

Register plugin

unregister()

Unregister plugin

register_hooks()

Register plugin hooks

unregister_hooks()

Unregister plugin hooks

abstract create_actions()

Create actions

datalab.plugins.**discover_plugins**() → list[type[*PluginBase*]]

Discover plugins using naming convention

Returns
List of discovered plugins (as classes)

datalab.plugins.**discover_v020_plugins**() → list[tuple[str, str]]

Discover v0.20 plugins (with `cdl_` prefix) without importing them

Returns
List of tuples (plugin_name, directory_path) for discovered v0.20 plugins

datalab.plugins.**get_available_plugins**() → list[*PluginBase*]

Instantiate and get available plugins

Returns
List of available plugins (as instances)

2.5.2 Migrating from DataLab v0.20 to v1.0

Warning: Critical compatibility notice:

DataLab v1.0 introduces **major breaking changes** that are **not backward compatible** with v0.20.

- **Plugins** developed for v0.20 **will not work** and must be updated
- **API changes** require code modifications for all custom integrations

Please read this guide carefully before upgrading to v1.0.

DataLab v1.0 introduces significant architectural changes compared to v0.20. The most important change is the **externalization of signal and image processing features** into a separate library called **Sigima**.

This architectural change improves modularity, testability, and enables the reuse of processing functions in other projects. However, it requires updates to existing plugins to adapt to the new API.

This guide describes the steps to migrate plugins from DataLab v0.20 to v1.0.

- *What changed in v1.0?*
  - *Major architectural changes*

- *Package renaming*
- *New result objects*
- *Unified processor interface*
- *Updating the imports*
  - *Import path changes*
- *Migrating plugin code*
  - *Example 1: Simple processing plugin*
  - *Example 2: Plugin using built-in features*
  - *Example 3: Plugin creating objects and handling results*
- *Working with result objects*
  - *Changes to result objects*
  - *Using TableResult and GeometryResult*
  - *Retrieving results from metadata*
- *Processor method changes*
  - *Using run_feature()*
  - *Method renaming*
  - *Built-in features using run_feature*
- *Testing your plugin*
- *Common issues and solutions*
  - *Import errors*
  - *Attribute errors on processor*
  - *Result object incompatibility*
  - *Missing parameters*
- *Additional resources*
- *Getting help*

What changed in v1.0?

Major architectural changes

The most significant change in DataLab v1.0 is the creation of the **Sigima library**, which now contains:

- **Signal and image objects** (`SignalObj`, `ImageObj`, ROI classes)
- **Processing parameters** (all `*Param` classes)
- **Processing functions** (`sigima.proc.signal` and `sigima.proc.image` modules)
- **Algorithms** for signal and image processing, or data type conversion, coordinates transformation, … (`sigima.tools` module)
- **I/O functions** (reading and writing signals/images)

- **Test data utilities** (test signal and image generation)
- **Result objects** (`TableResult` and `GeometryResult`, replacing `ResultProperties` and `ResultShape`)

What remains in DataLab:

- **GUI components** (panels, widgets, plot integration)
- **Plugin system** (`datalab.plugins` module)
- **Application logic** (main window, configuration, project management)
- **Result metadata adapters** (`TableAdapter`, `GeometryAdapter` for storing results in object metadata)

Package renaming

The DataLab package was renamed from `cdl` to `datalab` for clarity:

```
# v0.20
import cdl.obj
import cdl.param
from cdl.plugins import PluginBase

# v1.0
import sigima.objects
import sigima.params
from datalab.plugins import PluginBase
```

New result objects

DataLab v1.0 introduces two new immutable result types:

- **TableResult**: Replaces `ResultProperties` for tabular scalar results (e.g., statistics, measurements)
- **GeometryResult**: Replaces `ResultShape` for geometric results (e.g., detected features, contours)

These new result types are computation-oriented and free of application-specific logic (e.g., Qt, metadata). All metadata-related behaviors have been migrated to the DataLab application layer using **adapter classes** (`TableAdapter`, `GeometryAdapter`).

Unified processor interface

The processor methods have been unified and renamed:

- Most specific `compute_*` methods now use the generic `run_feature()` method
- Naming conventions updated: `compute_11` → `compute_1_to_1`, `compute_10` → `compute_1_to_0`, etc.

Updating the imports

Import path changes

The following table gives the equivalence between DataLab v0.20 and v1.0 imports.

The table below shows the mapping between DataLab v0.20 and v1.0 API. For most entries, only the import statement needs to be updated.

Table 6: DataLab v0.20 to v1.0 Compatibility Table

DataLab v0.20	DataLab v1.0
Module/package renames	
`cdl`	`datalab`
`cdl.obj`	`sigima.objects`
`cdl.param`	`sigima.params`
`cdl.computation.image`	`sigima.proc.image`
`cdl.computation.signal`	`sigima.proc.signal`
`cdl.plugins`	`datalab.plugins`
Object creation and manipulation	
`cdl.obj.create_image()`	`sigima.objects.create_image()`
`cdl.obj.create_signal()`	`sigima.objects.create_signal()`
`cdl.obj.ImageObj`	`sigima.objects.ImageObj`
`cdl.obj.SignalObj`	`sigima.objects.SignalObj`
`cdl.obj.create_image_roi()`	`sigima.objects.create_image_roi()`
`cdl.obj.create_signal_roi()`	`sigima.objects.create_signal_roi()`
`cdl.obj.ImageROI`	`sigima.objects.ImageROI`
`cdl.obj.SignalROI`	`sigima.objects.SignalROI`
Parameter classes	
`cdl.param.BinningParam`	`sigima.params.BinningParam`
`cdl.param.MovingMedianParam`	`sigima.params.MovingMedianParam`
`cdl.param.BlobOpenCVParam`	`sigima.params.BlobOpenCVParam`
`cdl.param.GaussianParam`	`sigima.params.GaussianParam`
`cdl.param.*Param`	`sigima.params.*Param`
Computation function wrappers	
`cdl.computation.image.Wrap11Func`	`sigima.proc.image.Wrap1to1Func`
`cdl.computation.signal.Wrap11Func`	`sigima.proc.signal.Wrap1to1Func`
`cdl.computation.image.dst_11()`	`sigima.proc.image.dst_1to1()`
`cdl.computation.signal.dst_11()`	`sigima.proc.signal.dst_1to1()`
Processor methods (*p* is a signal or image processor instance)	
`p.compute_11(...)`	`p.compute_1_to_1(...)`
`p.compute_10(...)`	`p.compute_1_to_0(...)`
`p.compute_n1(...)`	`p.compute_n_to_1(...)`
`p.compute_binning(param)`	`p.run_feature("binning", param)`
`p.compute_moving_median(param)`	`p.run_feature("moving_median", param)`
`p.compute_blob_opencv(param)`	`p.run_feature("blob_opencv", param)`
`p.compute_*(param)`	`p.run_feature("feature_name", param)`
Result objects	
`cdl.obj.ResultProperties`	`sigima.objects.TableResult`
`cdl.obj.ResultShape`	`sigima.objects.GeometryResult`
`result.add_to(obj)`	`TableAdapter(result).add_to(obj)`
`ResultShape.from_metadata_entry()`	`GeometryAdapter.from_metadata_entry()`

continues on next page

Table 6 – continued from previous page

DataLab v0.20	DataLab v1.0
`result.to_html(obj)`	`ResultHtmlGenerator.generate_html(result, obj)`
Test data functions	
`cdl.tests.data.*`	`sigima.tests.data.*`
`cdl.tests.data.create_paracetamol_signal()`	`sigima.tests.data.create_paracetamol_signal()`
`cdl.tests.data.add_gaussian_noise_to_signal(`	`sigima.tests.data.add_gaussian_noise_to_signal()`
`cdl.tests.data.create_noisygauss_image()`	`sigima.tests.data.create_noisy_gaussian_image()`
`cdl.tests.data.create_multigauss_image()`	`sigima.tests.data.create_multigaussian_image()`
New image parameters	
`edit_new_image_parameters(hide_image_dtype=T`	`edit_new_image_parameters(hide_dtype=True)`
`edit_new_image_parameters(hide_image_type=Tr`	`edit_new_image_parameters(hide_type=True)`
`edit_new_image_parameters(hide_image_height=`	`edit_new_image_parameters(hide_height=True)`
I/O functions	
`cdl.io.image.read_image()`	`sigima.io.image.read_image()`
`cdl.io.image.write_image()`	`sigima.io.image.write_image()`
`cdl.io.signal.read_signal()`	`sigima.io.signal.read_signal()`
`cdl.io.signal.write_signal()`	`sigima.io.signal.write_signal()`
Enumerations	
`cdl.obj.ImageDatatypes`	`sigima.objects.ImageDatatypes`
`cdl.obj.ImageTypes`	`sigima.objects.ImageTypes`
DataLab-specific features	
`cdl.config._`	`datalab.config._`
`cdl.plugins.PluginBase`	`datalab.plugins.PluginBase`
`cdl.plugins.PluginInfo`	`datalab.plugins.PluginInfo`

Migrating plugin code

Example 1: Simple processing plugin

Here's how a simple custom filter plugin needs to be updated:

DataLab v0.20:

```
import numpy as np
import scipy.ndimage as spi

import cdl.computation.image as cpi
import cdl.obj
import cdl.param
import cdl.plugins


def weighted_average_denoise(data: np.ndarray) -> np.ndarray:
    """Apply a custom denoising filter to an image."""
    def filter_func(values: np.ndarray) -> float:
        central_pixel = values[len(values) // 2]
```

(continues on next page)

(continued from previous page)

```
        differences = np.abs(values - central_pixel)
        weights = np.exp(-differences / np.mean(differences))
        return np.average(values, weights=weights)
    return spi.generic_filter(data, filter_func, size=5)


class CustomFilters(cdl.plugins.PluginBase):
    """DataLab Custom Filters Plugin"""

    PLUGIN_INFO = cdl.plugins.PluginInfo(
        name="My custom filters",
        version="1.0.0",
        description="This is an example plugin",
    )

    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        proc = self.imagepanel.processor
        with acth.new_menu(self.PLUGIN_INFO.name):
            for name, func in (("Weighted average denoise", weighted_average_denoise),):
                # Wrap function to handle ImageObj objects
                wrapped_func = cpi.Wrap11Func(func)
                acth.new_action(
                    name, triggered=lambda: proc.compute_11(wrapped_func, title=name)
                )
```

DataLab v1.0:

```
import numpy as np
import scipy.ndimage as spi
import sigima.proc.image as sipi

import datalab.plugins


def weighted_average_denoise(data: np.ndarray) -> np.ndarray:
    """Apply a custom denoising filter to an image."""
    def filter_func(values: np.ndarray) -> float:
        central_pixel = values[len(values) // 2]
        differences = np.abs(values - central_pixel)
        weights = np.exp(-differences / np.mean(differences))
        return np.average(values, weights=weights)
    return spi.generic_filter(data, filter_func, size=5)


class CustomFilters(datalab.plugins.PluginBase):
    """DataLab Custom Filters Plugin"""

    PLUGIN_INFO = datalab.plugins.PluginInfo(
        name="My custom filters",
        version="1.0.0",
```

(continues on next page)

(continued from previous page)

```
        description="This is an example plugin",
    )

    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        proc = self.imagepanel.processor
        with acth.new_menu(self.PLUGIN_INFO.name):
            for name, func in (("Weighted average denoise", weighted_average_denoise),):
                # Wrap function to handle ImageObj objects
                wrapped_func = sipi.Wrap1to1Func(func)
                acth.new_action(
                    name,
                    triggered=lambda: proc.compute_1_to_1(wrapped_func, title=name),
                )
```

Key changes:

1. `cdl.computation.image` → `sigima.proc.image`
2. `cdl.plugins` → `datalab.plugins`
3. `Wrap11Func` → `Wrap1to1Func`
4. `compute_11` → `compute_1_to_1`

Example 2: Plugin using built-in features

DataLab v0.20:

```
import cdl.obj
import cdl.param
import cdl.plugins


class ExtractBlobs(cdl.plugins.PluginBase):
    """DataLab Example Plugin"""

    PLUGIN_INFO = cdl.plugins.PluginInfo(
        name="Extract blobs (example)",
        version="1.0.0",
        description="This is an example plugin",
    )

    def preprocess(self) -> None:
        """Preprocess image"""
        panel = self.imagepanel
        param = cdl.param.BinningParam.create(sx=2, sy=2)
        panel.processor.compute_binning(param)
        panel.processor.compute_moving_median(cdl.param.MovingMedianParam.create(n=5))

    def detect_blobs(self) -> None:
        """Detect circular blobs"""
```

(continues on next page)

(continued from previous page)

```
        panel = self.imagepanel
        param = cdl.param.BlobOpenCVParam()
        param.filter_by_color = False
        param.min_area = 600.0
        param.max_area = 6000.0
        param.filter_by_circularity = True
        param.min_circularity = 0.8
        param.max_circularity = 1.0
        panel.processor.compute_blob_opencv(param)

    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        with acth.new_menu(self.PLUGIN_INFO.name):
            acth.new_action("Preprocess image", triggered=self.preprocess)
            acth.new_action("Detect circular blobs", triggered=self.detect_blobs)
```

DataLab v1.0:

```
import sigima.objects
import sigima.params

import datalab.plugins


class ExtractBlobs(datalab.plugins.PluginBase):
    """DataLab Example Plugin"""

    PLUGIN_INFO = datalab.plugins.PluginInfo(
        name="Extract blobs (example)",
        version="1.0.0",
        description="This is an example plugin",
    )

    def preprocess(self) -> None:
        """Preprocess image"""
        panel = self.imagepanel
        param = sigima.params.BinningParam.create(sx=2, sy=2)
        panel.processor.run_feature("binning", param)
        panel.processor.run_feature(
            "moving_median", sigima.params.MovingMedianParam.create(n=5)
        )

    def detect_blobs(self) -> None:
        """Detect circular blobs"""
        panel = self.imagepanel
        param = sigima.params.BlobOpenCVParam()
        param.filter_by_color = False
        param.min_area = 600.0
        param.max_area = 6000.0
        param.filter_by_circularity = True
        param.min_circularity = 0.8
```

(continues on next page)

(continued from previous page)

```
        param.max_circularity = 1.0
        panel.processor.run_feature("blob_opencv", param)

    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        with acth.new_menu(self.PLUGIN_INFO.name):
            acth.new_action("Preprocess image", triggered=self.preprocess)
            acth.new_action("Detect circular blobs", triggered=self.detect_blobs)
```

Key changes:

1. `cdl.param` → `sigima.params`
2. `cdl.plugins` → `datalab.plugins`
3. `compute_binning(param)` → `run_feature("binning", param)`
4. `compute_moving_median(param)` → `run_feature("moving_median", param)`
5. `compute_blob_opencv(param)` → `run_feature("blob_opencv", param)`

Example 3: Plugin creating objects and handling results

DataLab v0.20:

```
import numpy as np
import cdl.obj as dlo
import cdl.tests.data as test_data
from cdl.computation import image as cpima
from cdl.config import _
from cdl.plugins import PluginBase, PluginInfo


def add_noise_to_image(src: dlo.ImageObj, p: dlo.NormalRandomParam) -> dlo.ImageObj:
    """Add gaussian noise to image"""
    dst = cpima.dst_11(src, "add_gaussian_noise", f"mu={p.mu},sigma={p.sigma}")
    test_data.add_gaussian_noise_to_image(dst, p)
    return dst


class PluginTestData(PluginBase):
    """DataLab Test Data Plugin"""

    PLUGIN_INFO = PluginInfo(
        name=_("Test data"),
        version="1.0.0",
        description=_("Testing DataLab functionalities"),
    )

    def add_noise_to_image(self) -> None:
        """Add noise to image"""
        self.imagepanel.processor.compute_11(
            add_noise_to_image,
```

(continues on next page)

(continued from previous page)

```
            paramclass=dlo.NormalRandomParam,
            title=_("Add noise"),
        )

    def create_noisygauss_image(self) -> None:
        """Create 2D noisy gauss image"""
        newparam = self.edit_new_image_parameters(
            hide_image_height=True, hide_image_type=True
        )
        if newparam is not None:
            obj = test_data.create_noisygauss_image(newparam, add_annotations=False)
            self.proxy.add_object(obj)

    def create_actions(self) -> None:
        """Create actions"""
        iah = self.imagepanel.acthandler
        with iah.new_menu(_("Test data")):
            iah.new_action(_("Add noise to image"), triggered=self.add_noise_to_image)
            iah.new_action(
                _("Create 2D noisy gauss image"),
                triggered=self.create_noisygauss_image,
                select_condition="always",
            )
```

DataLab v1.0:

```
import numpy as np
import sigima.objects
import sigima.tests.data as test_data
from sigima.proc import image as sipi
from datalab.config import _
from datalab.plugins import PluginBase, PluginInfo


def add_noise_to_image(
    src: sigima.objects.ImageObj, p: sigima.objects.NormalDistribution2DParam
) -> sigima.objects.ImageObj:
    """Add gaussian noise to image"""
    dst = sipi.dst_1to1(src, "add_gaussian_noise", f"mu={p.mu},sigma={p.sigma}")
    test_data.add_gaussian_noise_to_image(dst, p)
    return dst


class PluginTestData(PluginBase):
    """DataLab Test Data Plugin"""

    PLUGIN_INFO = PluginInfo(
        name=_("Test data"),
        version="1.0.0",
        description=_("Testing DataLab functionalities"),
    )
```

(continues on next page)

(continued from previous page)

```
    def add_noise_to_image(self) -> None:
        """Add noise to image"""
        self.imagepanel.processor.compute_1_to_1(
            add_noise_to_image,
            paramclass=sigima.objects.NormalDistribution2DParam,
            title=_("Add noise"),
        )

    def create_noisy_gaussian_image(self) -> None:
        """Create 2D noisy gauss image"""
        newparam = self.edit_new_image_parameters(
            hide_height=True, hide_type=True
        )
        if newparam is not None:
            obj = test_data.create_noisy_gaussian_image(newparam, add_annotations=False)
            self.proxy.add_object(obj)

    def create_actions(self) -> None:
        """Create actions"""
        iah = self.imagepanel.acthandler
        with iah.new_menu(_("Test data")):
            iah.new_action(_("Add noise to image"), triggered=self.add_noise_to_image)
            iah.new_action(
                _("Create 2D noisy gaussian image"),
                triggered=self.create_noisy_gaussian_image,
                select_condition="always",
            )
```

Key changes:

1. `cdl.obj` → `sigima.objects`
2. `cdl.tests.data` → `sigima.tests.data`
3. `cdl.computation.image` → `sigima.proc.image`
4. `dst_11` → `dst_1to1`
5. `compute_11` → `compute_1_to_1`
6. `dlo.NormalRandomParam` → `sigima.objects.NormalDistribution2DParam`
7. `create_noisygauss_image` → `create_noisy_gaussian_image`
8. `hide_image_height` → `hide_height`, `hide_image_type` → `hide_type`

Working with result objects

Changes to result objects

The result object architecture has been completely redesigned in v1.0:

v0.20 Result Objects:

- `ResultProperties`: For tabular results (statistics, etc.)
- `ResultShape`: For geometric results (detected features, etc.)

- These objects contained Qt-specific code and metadata handling logic
- Methods like `add_to(obj)` and `from_metadata_entry()` were part of the result class

v1.0 Result Objects:

- `TableResult`: Immutable, computation-oriented table results
- `GeometryResult`: Immutable, computation-oriented geometry results
- No Qt or metadata dependencies
- Metadata handling moved to DataLab adapter classes

Using TableResult and GeometryResult

In v1.0, when your plugin receives results from `compute_1_to_0` operations, you'll work with `TableResult` or `GeometryResult` objects.

Example 1: Computing signal dynamic parameters (TableResult)

This example shows how to compute dynamic parameters (ENOB, SINAD, THD, etc.) and work with the resulting `TableResult`:

```
import sigima.params
import sigima.proc.signal
from datalab.adapters_metadata import TableAdapter

# Get the current signal from the panel
panel = self.signalpanel
obj = panel.objview.get_current_object()

# Compute dynamic parameters
param = sigima.params.DynamicParam.create(full_scale=1.0)
table = sigima.proc.signal.dynamic_parameters(obj, param)

# Access specific values from the table
enob = table["enob"][0]  # Effective Number of Bits
sinad = table["sinad"][0]  # Signal-to-Noise and Distortion Ratio
thd = table["thd"][0]  # Total Harmonic Distortion

# Convert to dictionary for easy access
result_dict = table.as_dict()
print(f"ENOB: {result_dict['enob']}")
print(f"SINAD: {result_dict['sinad']} dB")

# Store result in object metadata using adapter
adapter = TableAdapter(table)
adapter.add_to(obj)

# Convert to pandas DataFrame for further processing
df = table.to_dataframe()
print(df)
```

Example 2: Detecting image peaks (GeometryResult)

This example shows how to detect peaks in an image and work with the resulting `GeometryResult`:

```
import sigima.params
import sigima.proc.image
from datalab.adapters_metadata import GeometryAdapter

# Get the current image from the panel
panel = self.imagepanel
obj = panel.objview.get_current_object()

# Configure peak detection parameters
param = sigima.params.Peak2DDetectionParam.create(
    size=50,  # Neighborhood size
    threshold=0.5,  # Relative threshold
    create_rois=True  # Create ROI for each peak
)

# Compute peak detection
geometry = sigima.proc.image.peak_detection(obj, param)

# Access geometry data
coords = geometry.coords  # numpy array of shape (n_peaks, 2)
n_peaks = len(coords)
print(f"Detected {n_peaks} peaks")

# Iterate over detected peaks
for i, (x, y) in enumerate(coords):
    print(f"Peak {i+1}: x={x:.2f}, y={y:.2f}")

# Check geometry kind
from sigima.objects import KindShape
assert geometry.kind == KindShape.POINT

# Store geometry in object metadata using adapter
adapter = GeometryAdapter(geometry)
adapter.add_to(obj)
```

Example 3: Computing bandwidth (GeometryResult with segments)

This example shows how to compute signal bandwidth, which returns a segment geometry:

```
import sigima.proc.signal
from datalab.adapters_metadata import GeometryAdapter

# Get the current signal from the panel
panel = self.signalpanel
obj = panel.objview.get_current_object()

# Compute -3dB bandwidth
geometry = sigima.proc.signal.bandwidth_3db(obj)

# Access segment coordinates [x0, y0, x1, y1]
x0, y0, x1, y1 = geometry.coords[0]
print(f"Bandwidth segment: ({x0}, {y0}) to ({x1}, {y1})")
```

(continues on next page)

(continued from previous page)

```
# Calculate bandwidth length
bandwidth = geometry.segments_lengths()[0]
print(f"Bandwidth@-3dB: {bandwidth:.2f}")

# Store in metadata
adapter = GeometryAdapter(geometry)
adapter.add_to(obj)
```

Retrieving results from metadata

To retrieve results stored in object metadata:

```
from datalab.adapters_metadata import TableAdapter, GeometryAdapter

obj = ...  # Some signal or image object

# Iterate over all table results
for adapter in TableAdapter.iterate_from_obj(obj):
    result = adapter.result  # TableResult instance
    title = adapter.title
    df = result.to_dataframe()

# Iterate over all geometry results
for adapter in GeometryAdapter.iterate_from_obj(obj):
    result = adapter.result  # GeometryResult instance
    title = adapter.title
    coords = result.coords
```

Processor method changes

The processor interface has been unified in v1.0. Most specific `compute_*` methods now use the generic `run_feature()` method.

Using run_feature()

The `run_feature()` method is a unified interface for running processing features:

```
# v1.0 - Generic interface
proc = panel.processor

# Simple features (no parameters)
proc.run_feature("normalize")
proc.run_feature("fft")

# Features with parameters
param = sigima.params.GaussianParam.create(sigma=2.0)
proc.run_feature("gaussian_filter", param)

# Features with direct function reference
```

(continues on next page)

(continued from previous page)

```
import sigima.proc.image as sipi
proc.run_feature(sipi.normalize)
proc.run_feature(sipi.gaussian_filter, param)
```

Method renaming

Several processor methods have been renamed for consistency:

v0.20	v1.0
`compute_11(func, ...)`	`compute_1_to_1(func, ...)`
`compute_10(func, ...)`	`compute_1_to_0(func, ...)`
`compute_n1(func, ...)`	`compute_n_to_1(func, ...)`
`compute_1n(func, ...)`	`compute_1_to_n(func, ...)`
`compute_21(func, ...)`	`compute_2_to_1(func, ...)`

Built-in features using run_feature

Most built-in processing features that previously had dedicated `compute_*` methods now use `run_feature()`:

```
# v0.20
proc.compute_binning(param)
proc.compute_moving_median(param)
proc.compute_blob_opencv(param)
proc.compute_gaussian_filter(param)

# v1.0
proc.run_feature("binning", param)
proc.run_feature("moving_median", param)
proc.run_feature("blob_opencv", param)
proc.run_feature("gaussian_filter", param)
```

Some complex features still have dedicated methods:

```
# These still use dedicated methods in v1.0
proc.compute_roi_extraction(roi)
proc.compute_multigaussianfit()
proc.compute_peak_detection(param)
```

Testing your plugin

After migration, thoroughly test your plugin:

1. **Import checks**: Verify all imports work correctly
2. **Parameter creation**: Check that parameter classes are correctly instantiated
3. **Object creation**: Test signal/image object creation functions
4. **Processing operations**: Verify all processing features work as expected
5. **Result handling**: Check that results are correctly generated and stored

6. **GUI integration**: Test that menus and actions appear correctly

Common issues and solutions

Import errors

Problem: `ModuleNotFoundError: No module named 'cdl'`

Solution: Update all imports from `cdl.*` to either `datalab.*` or `sigima.*` depending on the module.

Attribute errors on processor

Problem: `AttributeError: 'ImageProcessor' object has no attribute 'compute_binning'`

Solution: Replace `compute_*` method calls with `run_feature()` calls:

```
# Wrong (v0.20 style)
proc.compute_binning(param)

# Correct (v1.0 style)
proc.run_feature("binning", param)
```

Result object incompatibility

Problem: `AttributeError: 'TableResult' object has no attribute 'add_to'`

Solution: Use adapter classes for metadata operations:

```
# Wrong (v0.20 style)
result.add_to(obj)

# Correct (v1.0 style)
from datalab.adapters_metadata import TableAdapter
TableAdapter(result).add_to(obj)
```

Missing parameters

Problem: `ImportError: cannot import name 'SomeParam' from 'datalab'`

Solution: Import parameters from `sigima.params` instead of `cdl.param`:

```
# Wrong
from cdl.param import GaussianParam

# Correct
from sigima.params import GaussianParam
```

Additional resources

- DataLab v1.0 documentation
- Sigima documentation
- Plugin examples
- *DataLab API reference*
- Sigima API reference

Getting help

If you encounter issues during migration:

1. Check the GitHub issue tracker
2. Consult the built-in plugin examples in the `datalab/plugins` directory

2.5.3 Macros

Overview

There are many ways to extend DataLab with new functionality (see *Plugins* or *Remote controlling*). The easiest way to do so is by using macros. Macros are small Python scripts that can be executed from the "Macro Panel" in DataLab.

Macros can be used to automate repetitive tasks, or to create new functionality. As the plugin and remote control system, macros rely on the DataLab high-level API to interact with the application. This means that you can reuse the same code snippets in macros, plugins, and remote control scripts.

> **Warning:** DataLab handles macros as Python scripts. This means that you can use the full power of Python to create your macros. Even though this is a powerful feature, it also means that you should be careful when running macros from unknown sources, as they can potentially harm your system.

See also:

The DataLab high-level API is documented in the *API* section. The plugin system is documented in the *Plugins* section, and the remote control system is documented in the *Remote controlling* section.

Main features

The Macro Panel is a simple interface to:

- Create new macros, using the "New macro" button.
- Rename existing macros, using the "Rename macro" button.
- Import/export macros from/to files, using the "Import macro" and "Export macro" buttons.
- Execute macros, using the "Run macro" button.
- Stop the execution of a macro, using the "Stop macro" button.

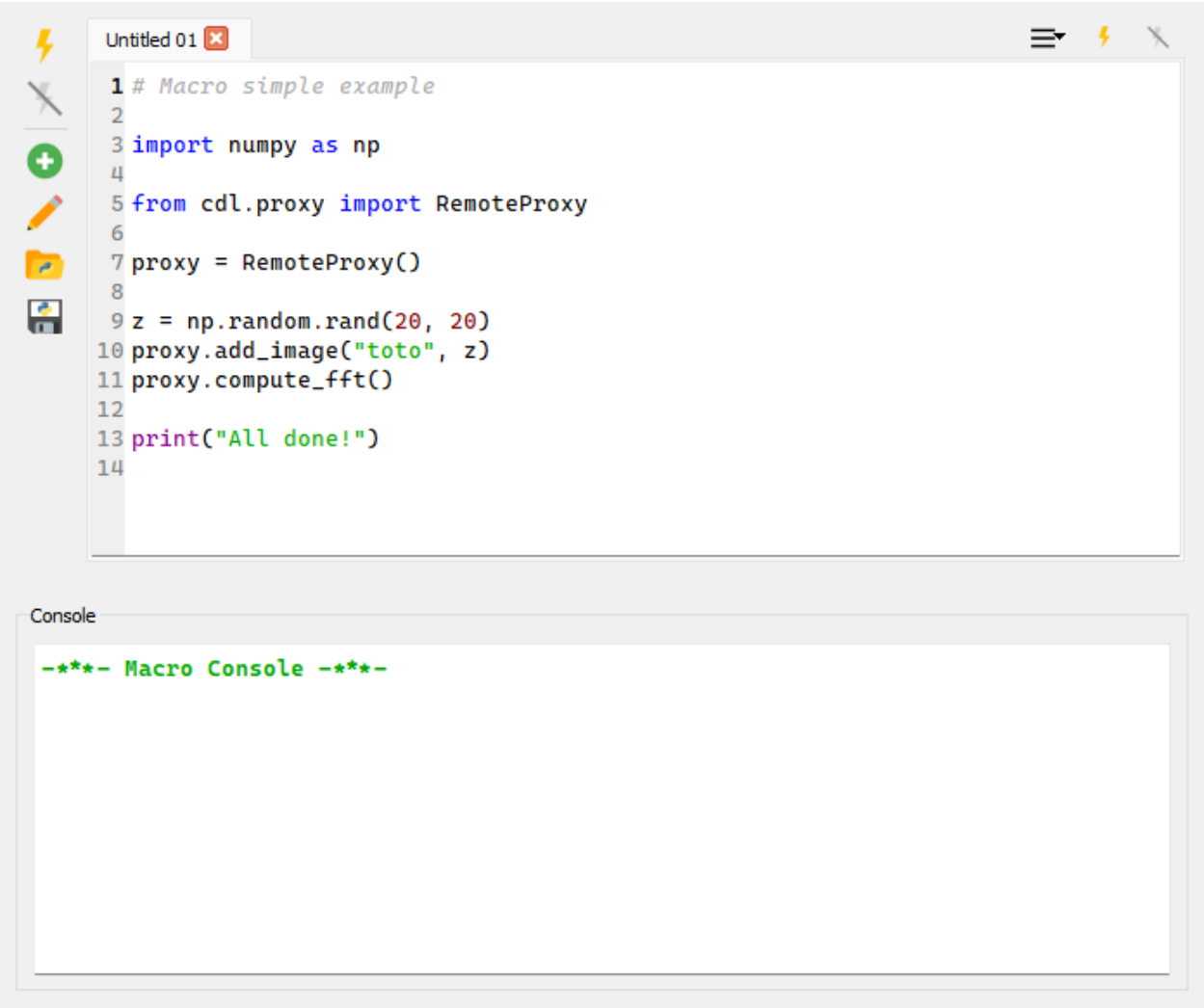


Fig. 70: The Macro Panel in DataLab.

Macros are embedded in the DataLab workspace, so they are saved together with the rest of the data (i.e. with signals and images) when exporting the workspace to a HDF5 file. This means that you can share your macros with other users simply by sharing the workspace file.

Note: Macro are executed in a separate process, so they won't block the main DataLab application. This means that you can continue working with DataLab while a macro is running and that *you can stop a macro at any time* using the button.

Example

For a detailed example of how to create a macro, see the *Prototyping a custom processing pipeline* tutorial.

2.5.4 Remote controlling

DataLab may be controlled remotely using the XML-RPC protocol which is natively supported by Python (and many other languages). Remote controlling allows to access DataLab main features from a separate process.

Note: If you are looking for a lighweight alternative solution to remote control DataLab (i.e. without having to install the whole DataLab package and its dependencies on your environment), you may use the Sigima package that provides a simple remote client for DataLab. To install it, just run: *pip install sigima*.

From an IDE

DataLab may be controlled remotely from an IDE (e.g. Spyder or any other IDE, or even a Jupyter Notebook) that runs a Python script. It allows to connect to a running DataLab instance, adds a signal and an image, and then runs calculations. This feature is exposed by the *RemoteProxy* class that is provided in module `datalab.control.proxy`.

From a third-party application

DataLab may also be controlled remotely from a third-party application, for the same purpose.

If the third-party application is written in Python 3, it may directly use the *RemoteProxy* class as mentioned above. From another language, it is also achievable, but it requires to implement a XML-RPC client in this language using the same methods of proxy server as in the *RemoteProxy* class.

Data (signals and images) may also be exchanged between DataLab and the remote client application, in both directions.

The remote client application may be written in any language that supports XML-RPC. For example, it is possible to write a remote client application in Python, Java, C++, C#, etc. The remote client application may be a graphical application or a command line application.

The remote client application may be run on the same computer as DataLab or on a different computer. In the latter case, the remote client application must know the IP address of the computer running DataLab.

The remote client application may be run before or after DataLab. In the latter case, the remote client application must try to connect to DataLab until it succeeds.

Supported features

Supported features are the following:

- Switch to signal or image panel
- Remove all signals and images
- Save current session to a HDF5 file
- Open HDF5 files into current session
- Browse HDF5 file
- Open a signal or an image from file
- Add a signal
- Add an image
- Get object list
- Run calculation with parameters

Note: The signal and image objects are described on this section: *Internal data model*.

Some examples are provided to help implementing such a communication between your application and DataLab:

- See module: `datalab.tests.features.control.remoteclient_app_test`
- See module: `datalab.tests.features.control.remoteclient_unit`

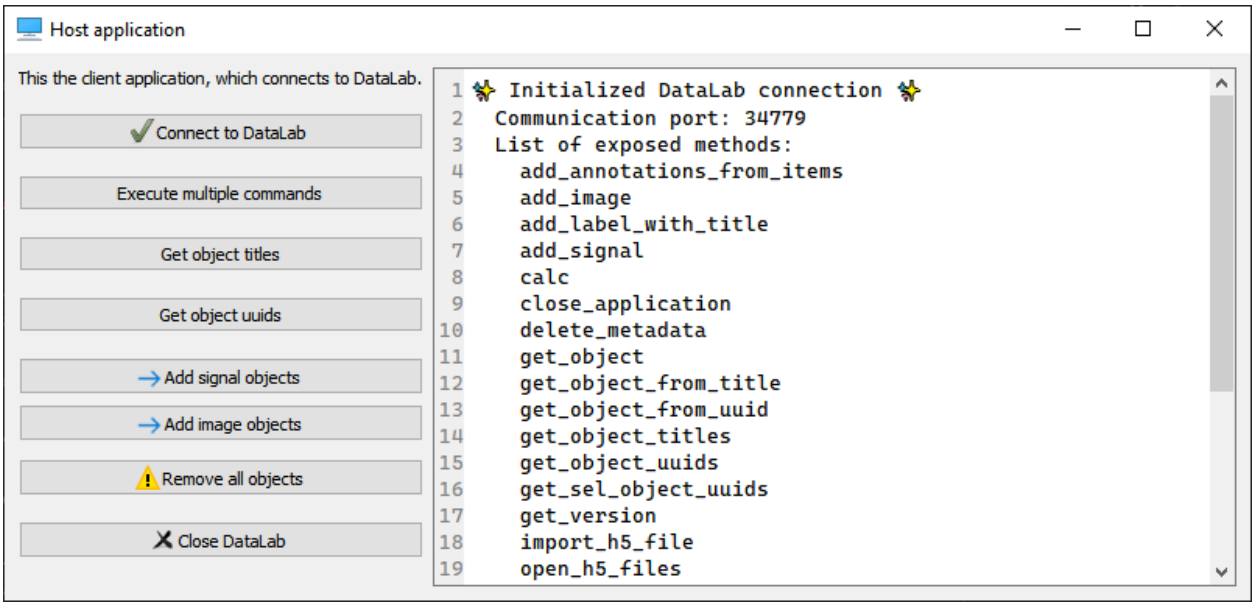


Fig. 71: Screenshot of remote client application test (`datalab.tests.features.control.remoteclient_app_test`)

Examples

When using Python 3, you may directly use the *RemoteProxy* class as in examples cited above or below.

Here is an example in Python 3 of a script that connects to a running DataLab instance, adds a signal and an image, and then runs calculations (the cell structure of the script make it convenient to be used in Spyder IDE):

```
"""
Example of remote control of DataLab current session,
from a Python script running outside DataLab (e.g. in Spyder)

Created on Fri May 12 12:28:56 2023

@author: p.raybaut
"""

# %% Importing necessary modules

# NumPy for numerical array computations:
import numpy as np

# DataLab remote control client:
from sigima.client import SimpleRemoteProxy as RemoteProxy

# %% Connecting to DataLab current session

proxy = RemoteProxy()

# %% Executing commands in DataLab (...)

z = np.random.rand(20, 20)
proxy.add_image("toto", z)

# %% Executing commands in DataLab (...)

proxy.toggle_auto_refresh(False)  # Turning off auto-refresh
x = np.array([1.0, 2.0, 3.0])
y = np.array([4.0, 5.0, -1.0])
proxy.add_signal("toto", x, y)

# %% Executing commands in DataLab (...)

proxy.calc("derivative")
proxy.toggle_auto_refresh(True)  # Turning on auto-refresh

# %% Executing commands in DataLab (...)

proxy.set_current_panel("image")

# %% Executing a lot of commands without refreshing DataLab

z = np.random.rand(400, 400)
proxy.add_image("foobar", z)
with proxy.context_no_refresh():
```

(continues on next page)

(continued from previous page)

```
    for _idx in range(100):
        proxy.calc("fft")
```

Here is a Python 2.7 reimplementation of this class:

```
# Copyright (c) DataLab Platform Developers, BSD 3-Clause license, see LICENSE file.

"""
DataLab remote controlling class for Python 2.7
"""

import io
import os
import os.path as osp
import socket
import sys

import ConfigParser as cp
import numpy as np
from guidata.userconfig import get_config_dir
from xmlrpclib import Binary, ServerProxy


def array_to_rpcbinary(data):
    """Convert NumPy array to XML-RPC Binary object, with shape and dtype"""
    dbytes = io.BytesIO()
    np.save(dbytes, data, allow_pickle=False)
    return Binary(dbytes.getvalue())


def get_datalab_xmlrpc_port():
    """Return DataLab current XML-RPC port"""
    if sys.platform == "win32" and "HOME" in os.environ:
        os.environ.pop("HOME")  # Avoid getting old WinPython settings dir
    fname = osp.join(get_config_dir(), ".DataLab", "DataLab.ini")
    ini = cp.ConfigParser()
    ini.read(fname)
    try:
        return ini.get("main", "rpc_server_port")
    except (cp.NoSectionError, cp.NoOptionError):
        raise ConnectionRefusedError("DataLab has not yet been executed")


class RemoteClient(object):
    """Object representing a proxy/client to DataLab XML-RPC server"""

    def __init__(self):
        self.port = None
        self.serverproxy = None

    def connect(self, port=None):
        """Connect to DataLab XML-RPC server"""
```

(continues on next page)

(continued from previous page)

```
        if port is None:
            port = get_datalab_xmlrpc_port()
        self.port = port
        url = "http://127.0.0.1:" + port
        self.serverproxy = ServerProxy(url, allow_none=True)
        try:
            self.get_version()
        except socket.error:
            raise ConnectionRefusedError("DataLab is currently not running")

    def get_version(self):
        """Return DataLab public version"""
        return self.serverproxy.get_version()

    def close_application(self):
        """Close DataLab application"""
        self.serverproxy.close_application()

    def raise_window(self):
        """Raise DataLab window"""
        self.serverproxy.raise_window()

    def get_current_panel(self):
        """Return current panel"""
        return self.serverproxy.get_current_panel()

    def set_current_panel(self, panel):
        """Switch to panel"""
        self.serverproxy.set_current_panel(panel)

    def reset_all(self):
        """Reset all application data"""
        self.serverproxy.reset_all()

    def toggle_auto_refresh(self, state):
        """Toggle auto refresh state"""
        self.serverproxy.toggle_auto_refresh(state)

    def toggle_show_titles(self, state):
        """Toggle show titles state"""
        self.serverproxy.toggle_show_titles(state)

    def save_to_h5_file(self, filename):
        """Save to a DataLab HDF5 file"""
        self.serverproxy.save_to_h5_file(filename)

    def open_h5_files(self, h5files, import_all, reset_all):
        """Open a DataLab HDF5 file or import from any other HDF5 file"""
        self.serverproxy.open_h5_files(h5files, import_all, reset_all)

    def import_h5_file(self, filename, reset_all):
        """Open DataLab HDF5 browser to Import HDF5 file"""
```

(continues on next page)

(continued from previous page)

```
        self.serverproxy.import_h5_file(filename, reset_all)

    def load_from_files(self, filenames):
        """Open objects from files in current panel (signals/images)"""
        self.serverproxy.load_from_files(filenames)

    def add_signal(
        self,
        title,
        xdata,
        ydata,
        xunit=None,
        yunit=None,
        xlabel=None,
        ylabel=None,
        group_id="",
        set_current=True,
    ):
        """Add signal data to DataLab"""
        xbinary = array_to_rpcbinary(xdata)
        ybinary = array_to_rpcbinary(ydata)
        p = self.serverproxy
        return p.add_signal(
            title, xbinary, ybinary, xunit, yunit, xlabel, ylabel, group_id, set_current
        )

    def add_image(
        self,
        title,
        data,
        xunit=None,
        yunit=None,
        zunit=None,
        xlabel=None,
        ylabel=None,
        zlabel=None,
        group_id="",
        set_current=True,
    ):
        """Add image data to DataLab"""
        zbinary = array_to_rpcbinary(data)
        p = self.serverproxy
        return p.add_image(
            title,
            zbinary,
            xunit,
            yunit,
            zunit,
            xlabel,
            ylabel,
            zlabel,
            group_id,
```

(continues on next page)

(continued from previous page)

```
            set_current,
        )

    def get_object_titles(self, panel=None):
        """Get object (signal/image) list for current panel"""
        return self.serverproxy.get_object_titles(panel)

    def get_object(self, nb_id_title=None, panel=None):
        """Get object (signal/image) by number, id or title"""
        return self.serverproxy.get_object(nb_id_title, panel)

    def get_object_uuids(self, panel=None, group=None):
        """Get object (signal/image) list for current panel"""
        return self.serverproxy.get_object_uuids(panel, group)


def test_remote_client():
    """DataLab Remote Client test"""
    datalab = RemoteClient()
    datalab.connect()
    data = np.array([[3, 4, 5], [7, 8, 0]], dtype=np.uint16)
    datalab.add_image("toto", data)


if __name__ == "__main__":
    test_remote_client()
```

Connection dialog

The DataLab package also provides a connection dialog that may be used to connect to a running DataLab instance. It is exposed by the `datalab.widgets.connection.ConnectionDialog` class.

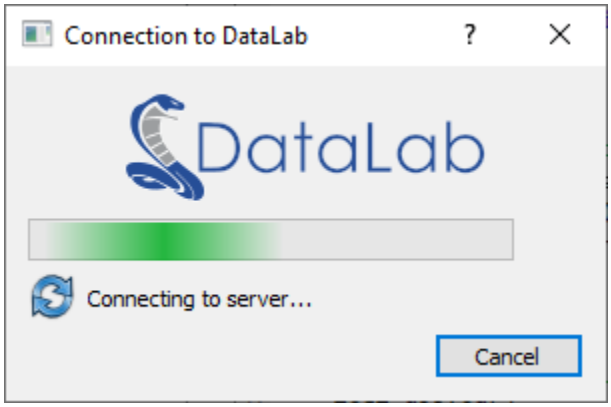


Fig. 72: Screenshot of connection dialog (`datalab.widgets.connection.ConnectionDialog`)

Example of use:

```
# Copyright (c) DataLab Platform Developers, BSD 3-Clause license, see LICENSE file.
```

(continues on next page)

(continued from previous page)

```
"""
DataLab Remote client connection dialog example
"""

# guitest: show,skip

from guidata.qthelpers import qt_app_context
from qtpy import QtWidgets as QW

from datalab.control.proxy import RemoteProxy
from datalab.widgets.connection import ConnectionDialog


def test_dialog():
    """Test connection dialog"""
    proxy = RemoteProxy(autoconnect=False)
    with qt_app_context():
        dlg = ConnectionDialog(proxy.connect)
        if dlg.exec():
            QW.QMessageBox.information(None, "Connection", "Successfully connected")
        else:
            QW.QMessageBox.critical(None, "Connection", "Connection failed")


if __name__ == "__main__":
    test_dialog()
```

Public API

class datalab.control.remote.**RemoteClient**

Object representing a proxy/client to DataLab XML-RPC server. This object is used to call DataLab functions from a Python script.

Examples

Here is a simple example of how to use RemoteClient in a Python script or in a Jupyter notebook:

```
>>> from datalab.remote import RemoteClient
>>> proxy = RemoteClient()
>>> proxy.connect()
Connecting to DataLab XML-RPC server...OK (port: 28867)
>>> proxy.get_version()
'1.0.0'
>>> proxy.add_signal("toto", np.array([1., 2., 3.]), np.array([4., 5., -1.]))
True
>>> proxy.get_object_titles()
['toto']
>>> proxy["toto"]
<sigima.objects.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1]
```

(continues on next page)

(continued from previous page)

```
<sigima.objects.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1].data
array([1., 2., 3.])
```

set_port(*port: str | None = None*) → None

Set XML-RPC port to connect to.

Parameters

port – XML-RPC port to connect to. If None, the port is automatically retrieved from DataLab configuration.

connect(*port: str | None = None, timeout: float | None = None, retries: int | None = None*) → None

Try to connect to DataLab XML-RPC server.

Parameters

- **port** – XML-RPC port to connect to. If not specified, the port is automatically retrieved from DataLab configuration.
- **timeout** – Maximum time to wait for connection in seconds. Defaults to 5.0. This is the total maximum wait time, not per retry.
- **retries** – Number of retries. Defaults to 10. This parameter is deprecated and will be removed in a future version (kept for backward compatibility).

Raises

- **ConnectionRefusedError** – Unable to connect to DataLab
- **ValueError** – Invalid timeout (must be >= 0.0)
- **ValueError** – Invalid number of retries (must be >= 1)

disconnect() → None

Disconnect from DataLab XML-RPC server.

is_connected() → bool

Return True if connected to DataLab XML-RPC server.

get_method_list() → list[str]

Return list of available methods.

add_signal(*title: str, xdata: ndarray, ydata: ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add signal data to DataLab.

Parameters

- **title** – Signal title
- **xdata** – X data
- **ydata** – Y data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **group_id** – group id in which to add the signal. Defaults to ""

- **set_current** – if True, set the added signal as current

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(*title: str, data: ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add image data to DataLab.

Parameters

- **title** – Image title
- **data** – Image data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **zunit** – Z unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **zlabel** – Z label. Defaults to ""
- **group_id** – group id in which to add the image. Defaults to ""
- **set_current** – if True, set the added image as current

Returns
True if image was added successfully, False otherwise

Raises
ValueError – Invalid data dtype

add_object(*obj: SignalObj | ImageObj, group_id: str = '', set_current: bool = True*) → None

Add object to DataLab.

Parameters

- **obj** – Signal or image object
- **group_id** – group id in which to add the object. Defaults to ""
- **set_current** – if True, set the added object as current

calc(*name: str, param: gds.DataSet | None = None*) → None

Call computation feature `name`

Note: This calls either the processor's `compute_<name>` method (if it exists), or the processor's `<name>` computation feature (if it is registered, using the `run_feature` method). It looks for the function in all panels, starting with the current one.

Parameters

- **name** – Compute function name

- **param** – Compute function parameter. Defaults to None.

Raises
ValueError – unknown function

get_object(*nb_id_title: int | str | None = None, panel: str | None = None*) → SignalObj | ImageObj

Get object (signal/image) from index.

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object).
- **panel** – Panel name. Defaults to None (current panel).

Returns
Object

Raises
KeyError – if object not found

get_object_shapes(*nb_id_title: int | str | None = None, panel: str | None = None*) → list

Get plot item shapes associated to object (signal/image).

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object).
- **panel** – Panel name. Defaults to None (current panel).

Returns
List of plot item shapes

add_annotations_from_items(*items: list, refresh_plot: bool = True, panel: str | None = None*) → None

Add object annotations (annotation plot items).

Parameters

- **items** – annotation plot items
- **refresh_plot** – refresh plot. Defaults to True.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.

add_group(*title: str, panel: str | None = None, select: bool = False*) → None

Add group to DataLab.

Parameters

- **title** – Group title
- **panel** – Panel name (valid values: "signal", "image"). Defaults to None.
- **select** – Select the group after creation. Defaults to False.

add_label_with_title(*title: str | None = None, panel: str | None = None*) → None

Add a label with object title on the associated plot

Parameters

- **title** – Label title. Defaults to None. If None, the title is the object title.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.

close_application() → None

Close DataLab application

context_no_refresh() → Generator[None, None, None]

Return a context manager to temporarily disable auto refresh.

Returns
Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.calc("fft")
...     proxy.calc("wiener")
...     proxy.calc("ifft")
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True, keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

get_current_panel() → str

Return current panel name.

Returns
"signal", "image", "macro"))

Return type
Panel name (valid values

get_group_titles_with_object_info() → tuple[list[str], list[list[str]], list[list[str]]]

Return groups titles and lists of inner objects uuids and titles.

Returns
groups titles, lists of inner objects uuids and titles

Return type
Tuple

get_object_titles(*panel: str | None = None*) → list[str]

Get object (signal/image) list for current panel. Objects are sorted by group number and object index in group.

Parameters
panel – panel name (valid values: "signal", "image", "macro"). If None, current data panel is used (i.e. signal or image panel).

Returns
List of object titles

Raises
ValueError – if panel not found

get_object_uuids(*panel: str | None = None, group: int | str | None = None*) → list[str]

Get object (signal/image) uuid list for current panel. Objects are sorted by group number and object index in group.

Parameters

- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.
- **group** – Group number, or group id, or group title. Defaults to None (all groups).

Returns
List of object uuids

Raises
ValueError – if panel not found

classmethod get_public_methods() → list[str]

Return all public methods of the class, except itself.

Returns
List of public methods

get_sel_object_uuids(*include_groups: bool = False*) → list[str]

Return selected objects uuids.

Parameters
include_groups – If True, also return objects from selected groups.

Returns
List of selected objects uuids.

get_version() → str

Return DataLab public version.

Returns
DataLab version

import_h5_file(*filename: str, reset_all: bool | None = None*) → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- **filename** – HDF5 file name
- **reset_all** – Reset all application data. Defaults to None.

import_macro_from_file(*filename: str*) → None

Import macro from file

Parameters
filename – Filename.

load_from_directory(*path: str*) → None

Open objects from directory in current panel (signals/images).

Parameters
path – directory path

load_from_files(*filenames: list[str]*) → None

Open objects from files in current panel (signals/images).

Parameters
filenames – list of file names

open_h5_files(*h5files: list[str] | None = None, import_all: bool | None = None, reset_all: bool | None = None*) → None

Open a DataLab HDF5 file or import from any other HDF5 file.

Parameters

- **h5files** – List of HDF5 files to open. Defaults to None.
- **import_all** – Import all objects from HDF5 files. Defaults to None.
- **reset_all** – Reset all application data. Defaults to None.

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

run_macro(*number_or_title: int | str | None = None*) → None

Run macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters
filename – HDF5 file name

select_groups(*selection: list[int | str] | None = None, panel: str | None = None*) → None

Select groups in current panel.

Parameters

- **selection** – List of group numbers (1 to N), or list of group uuids, or None to select all groups. Defaults to None.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

select_objects(*selection: list[int | str], panel: str | None = None*) → None

Select objects in current panel.

Parameters

- **selection** – List of object numbers (1 to N) or uuids to select
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

set_current_panel(*panel: str*) → None

Switch to panel.

Parameters
panel – Panel name (valid values: "signal", "image", "macro"))

stop_macro(*number_or_title: int | str | None = None*) → None

Stop macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters
state – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters
state – Show titles state

2.5.5 Internal data model

In its internal data model, DataLab stores data using two main classes:

- `sigima.objects.SignalObj`, which represents a signal object, and
- `sigima.objects.ImageObj`, which represents an image object.

These classes are defined in the `sigima.objects` package.

Also, DataLab uses many different datasets (based on guidata's `DataSet` class) to store the parameters of the computations. These datasets are defined in different modules but are exposed publicly in the `sigima.params` package.

See also:

The *API* section for more information on the public API.

2.5.6 Log viewer

Despite countless efforts (unit testing, test coverage…), DataLab might crash or behave unexpectedly.

For those situations, DataLab provides two logs (located in your home directory):

- "Traceback log", for Python exceptions
- "Faulthandler log", for system failures (e.g. Qt-related crash)

Note: The log viewer is accessible from the "?" menu in the main window.

If DataLab crashed or if any Python exception is raised during its execution, those log files will be updated accordingly. DataLab will even notify that new informations are available in log files at next startup. This is an invitation to submit a bug report.

Reporting unexpected behavior or any other bug on GitHub Issues will be greatly appreciated, especially if those log file contents are attached to the report (as information on your installation configuration, see *Installation and Configuration*).

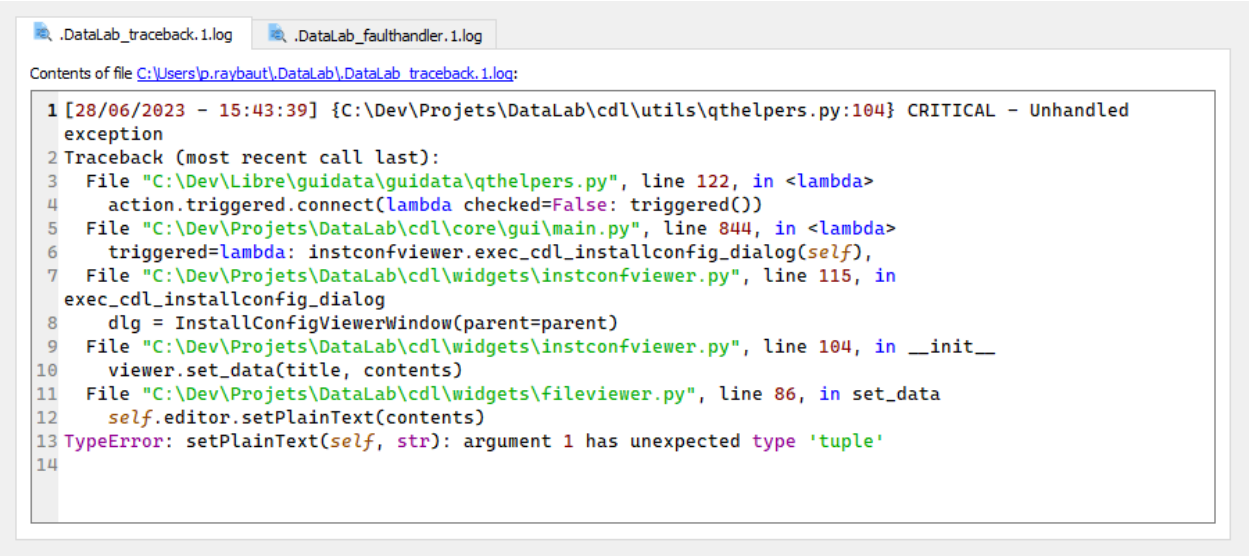


Fig. 73: DataLab log viewer (see "?" menu)

2.5.7 Installation and Configuration

The **Installation and Configuration** dialog box is a diagnostic tool available from DataLab's graphical interface. It provides a structured overview of the current installation, user-specific settings, and available plugins and I/O features.

This tool is primarily designed for troubleshooting and verification purposes — whether you are a user trying to understand how DataLab is set up on your system, or a developer validating your plugin integration.

To open the dialog box, go to the "?" menu in the main DataLab window and select **Installation and configuration**. This will open a dialog window that displays all relevant information about your DataLab installation.

The dialog window is divided into **three tabs**, each covering a specific aspect of the environment.

1. Installation Configuration

This tab displays system-level information about the DataLab installation, including:

- Application version;
- Platform and operating system;
- Whether DataLab is running in frozen (standalone) mode or as a standard Python package;
- Internal paths used for data storage and plugin discovery;
- Python interpreter details (version, architecture, paths);
- Location of the *datalab* package.

This section is especially useful for debugging path-related issues or confirming the installation context (e.g., standalone vs. pip-installed).

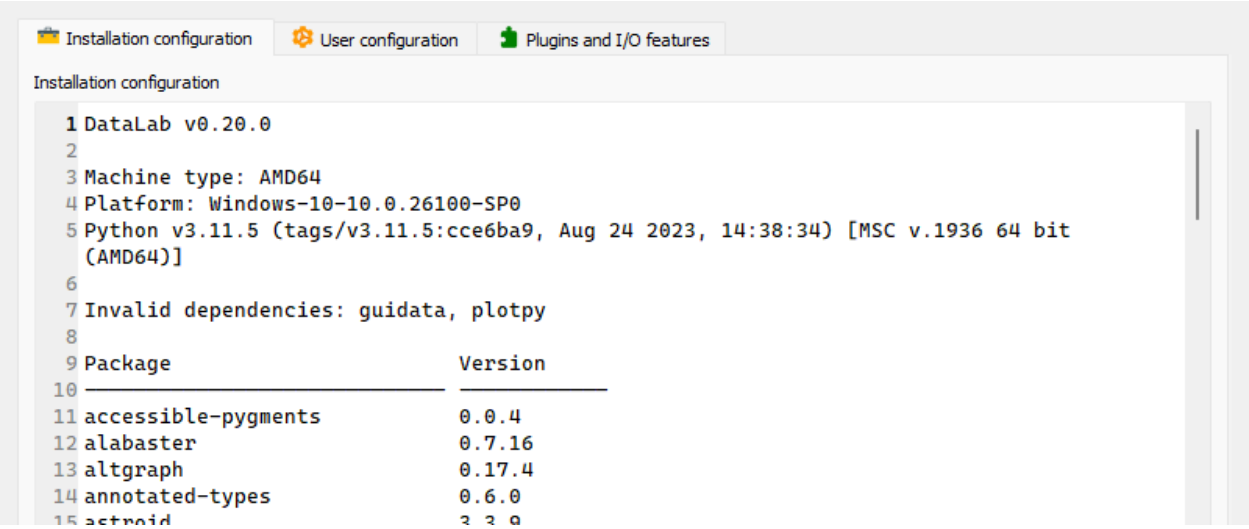


Fig. 74: Installation and configuration (see "?" menu), tab 1

2. User Configuration

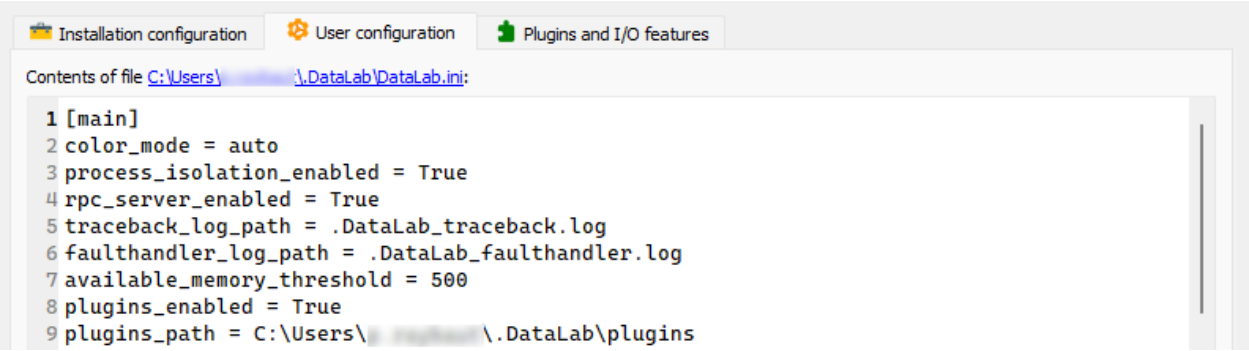


Fig. 75: Installation and configuration (see "?" menu), tab 2

This tab lists user-specific configuration details, through the contents of DataLab's configuration file.

It helps users verify their personalized settings and ensures that DataLab is reading the correct directories for user-defined plugins or preferences.

3. Plugins and I/O Features

This tab lists all plugins and I/O handlers currently available in the application.

This view is ideal for verifying that all expected plugins and I/O features are detected, loaded, and functional.

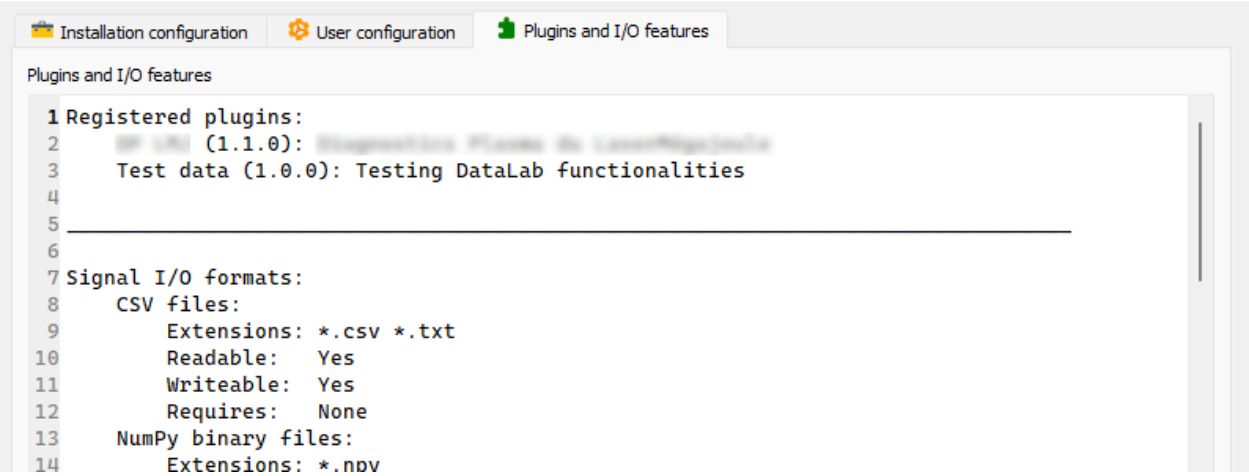


Fig. 76: Installation and configuration (see "?" menu), tab 3

Usage Tips

- You can copy text from any of the tabs for diagnostic or support purposes.
- If a plugin or I/O feature does not appear, check that its filename starts with *datalab_* and that it is located in a recognized plugin directory (see the *User Configuration* tab).

Note: Reporting unexpected behavior or any other bug on GitHub Issues will be greatly appreciated, especially if the contents of this viewer are attached to the report (as well as log files, see *Log viewer*).

2.5.8 Command line features

Run DataLab

To run DataLab from the command line, type the following:

```
$ datalab
```

To show help on command line usage, simply run:

```
$ datalab --help
usage: app.py [-h] [-b path] [-v] [--unattended] [--screenshot] [--delay DELAY] [--
↪xmlrpcport PORT]
              [--verbose {quiet,minimal,normal}]
              [h5]

Run DataLab

positional arguments:
h5                    HDF5 file names (separated by ';'), optionally with dataset name␣
↪(separated by ',')

options:
```

(continues on next page)

(continued from previous page)

```
-h, --help            show this help message and exit
-b path, --h5browser path
                        path to open with HDF5 browser
-v, --version         show DataLab version
--reset               reset DataLab configuration
--unattended          non-interactive mode
--accept_dialogs      accept dialogs in unattended mode
--screenshot          automatic screenshots
--delay DELAY         delay (ms) before quitting application in unattended mode
--xmlrpcport XMLRPCPORT
                        XML-RPC port number
--verbose {quiet,normal,debug}
                        verbosity level: for debugging/testing purpose
```

Open HDF5 file at startup

To open HDF5 files, or even import only a specified HDF5 dataset, use the following:

```
$ datalab /path/to/file1.h5
$ datalab /path/to/file1.h5,/path/to/dataset1
$ datalab /path/to/file1.h5,/path/to/dataset1;/path/to/file2.h5,/path/to/dataset2
```

Open HDF5 browser at startup

To open the HDF5 browser at startup, use one of the following commands:

```
$ datalab -b /path/to/file1.h5
$ datalab --h5browser /path/to/file1.h5
```

Run DataLab demo

To execute DataLab demo, run the following:

```
$ datalab-demo
```

Run technical validation tests

Note: Technical validation tests are directly included in individual unit tests and are disseminated throughout the code. The test functions including validation tests are marked with the `@pytest.mark.validation` decorator.

To execute DataLab technical validation tests, run the following:

```
$ pytest -m validation
```

See also:

See section *Overview & Common features* for more information on DataLab's validation strategy.

Run complete test suite

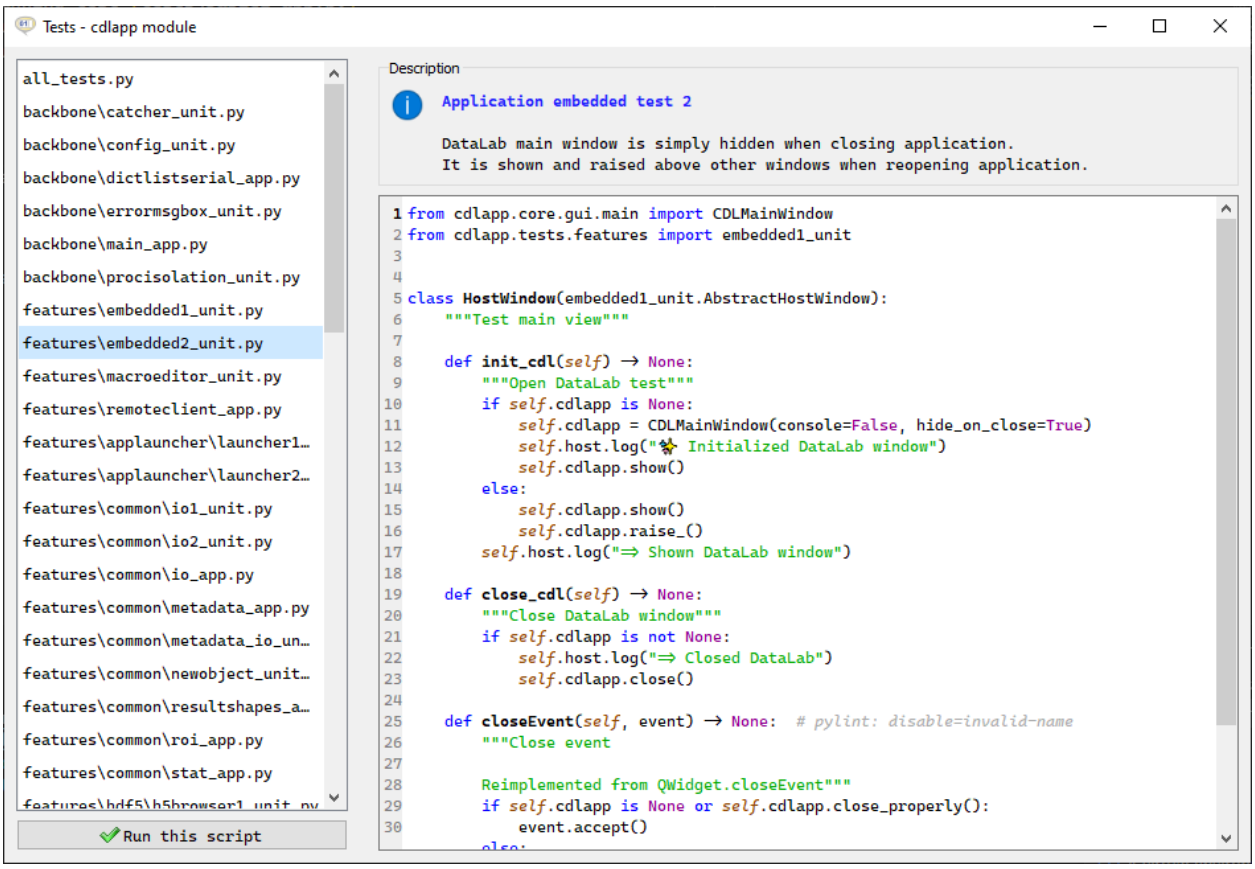
To execute all DataLab unit tests, simply run:

```
$ pytest
```

Run interactive tests

To execute DataLab interactive tests, run the following:

```
$ datalab-tests
```



2.5.9 API

The public Application Programming Interface (API) of DataLab offers a set of functions to access the DataLab features.

Note: For more details about the *sigima* package, please refer to the Sigima documentation. The Sigima API is especially useful to write scripts that can be run in DataLab (see *Macros*) or to develop plugins (see *Plugins*).

Proxy objects (datalab.control.proxy)

The *datalab.control.proxy* module provides a way to access DataLab features from a proxy class.

Remote proxy

The remote proxy is used when DataLab is started from a different process than the proxy. In this case, the proxy connects to DataLab XML-RPC server.

class datalab.control.proxy.**RemoteProxy**(*autoconnect: bool = True*)

DataLab remote proxy class.

This class provides access to DataLab features from a proxy class. This is the remote version of proxy, which is used when DataLab is started from a different process than the proxy.

Parameters

autoconnect – Automatically connect to DataLab XML-RPC server.

Raises

- **ConnectionRefusedError** – Unable to connect to DataLab
- **ValueError** – Invalid timeout (must be >= 0.0)
- **ValueError** – Invalid number of retries (must be >= 1)

Examples

Here is a simple example of how to use RemoteProxy in a Python script or in a Jupyter notebook:

```
>>> from datalab.control.proxy import RemoteProxy
>>> proxy = RemoteProxy()
Connecting to DataLab XML-RPC server...OK (port: 28867)
>>> proxy.get_version()
'1.0.0'
>>> proxy.add_signal("toto", np.array([1., 2., 3.]), np.array([4., 5., -1.]))
True
>>> proxy.get_object_titles()
['toto']
>>> proxy["toto"]  # from title
<sigima.objects.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1]  # from number
<sigima.objects.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1].data
array([1., 2., 3.])
>>> proxy.set_current_panel("image")
```

add_annotations_from_items(*items: list, refresh_plot: bool = True, panel: str | None = None*) → None

Add object annotations (annotation plot items).

Parameters

- **items** – annotation plot items
- **refresh_plot** – refresh plot. Defaults to True.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.

add_group(*title: str, panel: str | None = None, select: bool = False*) → None

Add group to DataLab.

Parameters

- **title** – Group title
- **panel** – Panel name (valid values: "signal", "image"). Defaults to None.
- **select** – Select the group after creation. Defaults to False.

add_image(*title: str, data: ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add image data to DataLab.

Parameters

- **title** – Image title
- **data** – Image data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **zunit** – Z unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **zlabel** – Z label. Defaults to ""
- **group_id** – group id in which to add the image. Defaults to ""
- **set_current** – if True, set the added image as current

Returns

True if image was added successfully, False otherwise

Raises

ValueError – Invalid data dtype

add_label_with_title(*title: str | None = None, panel: str | None = None*) → None

Add a label with object title on the associated plot

Parameters

- **title** – Label title. Defaults to None. If None, the title is the object title.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.

add_object(*obj: SignalObj | ImageObj, group_id: str = '', set_current: bool = True*) → None

Add object to DataLab.

Parameters

- **obj** – Signal or image object
- **group_id** – group id in which to add the object. Defaults to ""
- **set_current** – if True, set the added object as current

add_signal(*title: str, xdata: ndarray, ydata: ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add signal data to DataLab.

Parameters

- **title** – Signal title
- **xdata** – X data
- **ydata** – Y data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **group_id** – group id in which to add the signal. Defaults to ""
- **set_current** – if True, set the added signal as current

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

calc(*name: str*, *param: gds.DataSet | None = None*) → None

Call computation feature name

Note: This calls either the processor's compute_<name> method (if it exists), or the processor's <name> computation feature (if it is registered, using the run_feature method). It looks for the function in all panels, starting with the current one.

Parameters

- **name** – Compute function name
- **param** – Compute function parameter. Defaults to None.

Raises
ValueError – unknown function

close_application() → None

Close DataLab application

connect(*port: str | None = None*, *timeout: float | None = None*, *retries: int | None = None*) → None

Try to connect to DataLab XML-RPC server.

Parameters

- **port** – XML-RPC port to connect to. If not specified, the port is automatically retrieved from DataLab configuration.
- **timeout** – Maximum time to wait for connection in seconds. Defaults to 5.0. This is the total maximum wait time, not per retry.
- **retries** – Number of retries. Defaults to 10. This parameter is deprecated and will be removed in a future version (kept for backward compatibility).

Raises

- **ConnectionRefusedError** – Unable to connect to DataLab
- **ValueError** – Invalid timeout (must be >= 0.0)
- **ValueError** – Invalid number of retries (must be >= 1)

context_no_refresh() → Generator[None, None, None]

Return a context manager to temporarily disable auto refresh.

Returns
Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.calc("fft")
...     proxy.calc("wiener")
...     proxy.calc("ifft")
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True, keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

disconnect() → None

Disconnect from DataLab XML-RPC server.

get_current_panel() → str

Return current panel name.

Returns
"signal", "image", "macro"))

Return type
Panel name (valid values

get_group_titles_with_object_info() → tuple[list[str], list[list[str]], list[list[str]]]

Return groups titles and lists of inner objects uuids and titles.

Returns
groups titles, lists of inner objects uuids and titles

Return type
Tuple

get_method_list() → list[str]

Return list of available methods.

get_object(*nb_id_title: int | str | None = None, panel: str | None = None*) → SignalObj | ImageObj

Get object (signal/image) from index.

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object).
- **panel** – Panel name. Defaults to None (current panel).

Returns
Object

Raises
KeyError – if object not found

get_object_shapes(*nb_id_title: int | str | None = None, panel: str | None = None*) → list

Get plot item shapes associated to object (signal/image).

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object).
- **panel** – Panel name. Defaults to None (current panel).

Returns
List of plot item shapes

get_object_titles(*panel: str | None = None*) → list[str]

Get object (signal/image) list for current panel. Objects are sorted by group number and object index in group.

Parameters
panel – panel name (valid values: "signal", "image", "macro"). If None, current data panel is used (i.e. signal or image panel).

Returns
List of object titles

Raises
ValueError – if panel not found

get_object_uuids(*panel: str | None = None, group: int | str | None = None*) → list[str]

Get object (signal/image) uuid list for current panel. Objects are sorted by group number and object index in group.

Parameters

- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.
- **group** – Group number, or group id, or group title. Defaults to None (all groups).

Returns
List of object uuids

Raises
ValueError – if panel not found

classmethod get_public_methods() → list[str]

Return all public methods of the class, except itself.

Returns
List of public methods

get_sel_object_uuids(*include_groups: bool = False*) → list[str]

Return selected objects uuids.

Parameters
include_groups – If True, also return objects from selected groups.

Returns
List of selected objects uuids.

get_version() → str

Return DataLab public version.

Returns
DataLab version

import_h5_file(*filename: str*, *reset_all: bool | None = None*) → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- **filename** – HDF5 file name
- **reset_all** – Reset all application data. Defaults to None.

import_macro_from_file(*filename: str*) → None

Import macro from file

Parameters
filename – Filename.

is_connected() → bool

Return True if connected to DataLab XML-RPC server.

load_from_directory(*path: str*) → None

Open objects from directory in current panel (signals/images).

Parameters
path – directory path

load_from_files(*filenames: list[str]*) → None

Open objects from files in current panel (signals/images).

Parameters
filenames – list of file names

open_h5_files(*h5files: list[str] | None = None*, *import_all: bool | None = None*, *reset_all: bool | None = None*) → None

Open a DataLab HDF5 file or import from any other HDF5 file.

Parameters

- **h5files** – List of HDF5 files to open. Defaults to None.
- **import_all** – Import all objects from HDF5 files. Defaults to None.
- **reset_all** – Reset all application data. Defaults to None.

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

run_macro(*number_or_title: int | str | None = None*) → None

Run macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters
filename – HDF5 file name

select_groups(*selection: list[int | str] | None = None, panel: str | None = None*) → None

Select groups in current panel.

Parameters

- **selection** – List of group numbers (1 to N), or list of group uuids, or None to select all groups. Defaults to None.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

select_objects(*selection: list[int | str], panel: str | None = None*) → None

Select objects in current panel.

Parameters

- **selection** – List of object numbers (1 to N) or uuids to select
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

set_current_panel(*panel: str*) → None

Switch to panel.

Parameters
panel – Panel name (valid values: "signal", "image", "macro"))

set_port(*port: str | None = None*) → None

Set XML-RPC port to connect to.

Parameters
port – XML-RPC port to connect to. If None, the port is automatically retrieved from DataLab configuration.

stop_macro(*number_or_title: int | str | None = None*) → None

Stop macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters
state – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters
state – Show titles state

Local proxy

The local proxy is used when DataLab is started from the same process as the proxy. In this case, the proxy is directly connected to DataLab main window instance. The typical use case is high-level scripting.

class datalab.control.proxy.**LocalProxy**(*datalab: DLMainWindow | ServerProxy | None = None*)

DataLab local proxy class.

This class provides access to DataLab features from a proxy class. This is the local version of proxy, which is used when DataLab is started from the same process as the proxy.

Parameters
datalab (DLMainWindow) – DLMainWindow instance.

add_signal(*title: str, xdata: ndarray, ydata: ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add signal data to DataLab.

Parameters

- **title** – Signal title
- **xdata** – X data
- **ydata** – Y data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **group_id** – group id in which to add the signal. Defaults to ""
- **set_current** – if True, set the added signal as current

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(*title: str, data: ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add image data to DataLab.

Parameters

- **title** – Image title
- **data** – Image data
- **xunit** – X unit. Defaults to ""

- **yunit** – Y unit. Defaults to “”
- **zunit** – Z unit. Defaults to “”
- **xlabel** – X label. Defaults to “”
- **ylabel** – Y label. Defaults to “”
- **zlabel** – Z label. Defaults to “”
- **group_id** – group id in which to add the image. Defaults to “”
- **set_current** – if True, set the added image as current

Returns
True if image was added successfully, False otherwise

Raises
ValueError – Invalid data dtype

add_object(*obj: SignalObj | ImageObj*, *group_id: str = ''*, *set_current: bool = True*) → None

Add object to DataLab.

Parameters

- **obj** – Signal or image object
- **group_id** – group id in which to add the object. Defaults to “”
- **set_current** – if True, set the added object as current

calc(*name: str*, *param: gds.DataSet | None = None*) → None

Call computation feature `name`

Note: This calls either the processor’s `compute_<name>` method (if it exists), or the processor’s `<name>` computation feature (if it is registered, using the `run_feature` method). It looks for the function in all panels, starting with the current one.

Parameters

- **name** – Compute function name
- **param** – Compute function parameter. Defaults to None

Raises
ValueError – unknown function

get_object(*nb_id_title: int | str | None = None*, *panel: str | None = None*) → SignalObj | ImageObj

Get object (signal/image) from index.

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object)
- **panel** – Panel name. Defaults to None (current panel)

Returns
Object

Raises
KeyError – if object not found

get_object_shapes(*nb_id_title: int | str | None = None, panel: str | None = None*) → list

Get plot item shapes associated to object (signal/image).

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object)
- **panel** – Panel name. Defaults to None (current panel)

Returns
List of plot item shapes

add_annotations_from_items(*items: list, refresh_plot: bool = True, panel: str | None = None*) → None

Add object annotations (annotation plot items).

Parameters

- **itemsTrue** – annotation plot items
- **refresh_plotTrue** – refresh plot. Defaults to True
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used

add_group(*title: str, panel: str | None = None, select: bool = False*) → None

Add group to DataLab.

Parameters

- **title** – Group title
- **panel** – Panel name (valid values: "signal", "image"). Defaults to None.
- **select** – Select the group after creation. Defaults to False.

add_label_with_title(*title: str | None = None, panel: str | None = None*) → None

Add a label with object title on the associated plot

Parameters

- **title** – Label title. Defaults to None. If None, the title is the object title.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.

close_application() → None

Close DataLab application

context_no_refresh() → Generator[None, None, None]

Return a context manager to temporarily disable auto refresh.

Returns
Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.calc("fft")
...     proxy.calc("wiener")
...     proxy.calc("ifft")
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True, keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

get_current_panel() → str

Return current panel name.

Returns
"signal", "image", "macro"))

Return type
Panel name (valid values

get_group_titles_with_object_info() → tuple[list[str], list[list[str]], list[list[str]]]

Return groups titles and lists of inner objects uuids and titles.

Returns
groups titles, lists of inner objects uuids and titles

Return type
Tuple

get_object_titles(*panel: str | None = None*) → list[str]

Get object (signal/image) list for current panel. Objects are sorted by group number and object index in group.

Parameters
panel – panel name (valid values: "signal", "image", "macro"). If None, current data panel is used (i.e. signal or image panel).

Returns
List of object titles

Raises
ValueError – if panel not found

get_object_uuids(*panel: str | None = None, group: int | str | None = None*) → list[str]

Get object (signal/image) uuid list for current panel. Objects are sorted by group number and object index in group.

Parameters

- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used.
- **group** – Group number, or group id, or group title. Defaults to None (all groups).

Returns
List of object uuids

Raises
`ValueError` – if panel not found

`classmethod get_public_methods()` → list[str]

Return all public methods of the class, except itself.

Returns
List of public methods

`get_sel_object_uuids(include_groups: bool = False)` → list[str]

Return selected objects uuids.

Parameters
`include_groups` – If True, also return objects from selected groups.

Returns
List of selected objects uuids.

`get_version()` → str

Return DataLab public version.

Returns
DataLab version

`import_h5_file(filename: str, reset_all: bool | None = None)` → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- `filename` – HDF5 file name
- `reset_all` – Reset all application data. Defaults to None.

`import_macro_from_file(filename: str)` → None

Import macro from file

Parameters
`filename` – Filename.

`load_from_directory(path: str)` → None

Open objects from directory in current panel (signals/images).

Parameters
`path` – directory path

`load_from_files(filenames: list[str])` → None

Open objects from files in current panel (signals/images).

Parameters
`filenames` – list of file names

`open_h5_files(h5files: list[str] | None = None, import_all: bool | None = None, reset_all: bool | None = None)` → None

Open a DataLab HDF5 file or import from any other HDF5 file.

Parameters

- `h5files` – List of HDF5 files to open. Defaults to None.
- `import_all` – Import all objects from HDF5 files. Defaults to None.
- `reset_all` – Reset all application data. Defaults to None.

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

run_macro(*number_or_title: int | str | None = None*) → None

Run macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters
filename – HDF5 file name

select_groups(*selection: list[int | str] | None = None, panel: str | None = None*) → None

Select groups in current panel.

Parameters

- **selection** – List of group numbers (1 to N), or list of group uuids, or None to select all groups. Defaults to None.
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

select_objects(*selection: list[int | str], panel: str | None = None*) → None

Select objects in current panel.

Parameters

- **selection** – List of object numbers (1 to N) or uuids to select
- **panel** – panel name (valid values: "signal", "image"). If None, current panel is used. Defaults to None.

set_current_panel(*panel: str*) → None

Switch to panel.

Parameters
panel – Panel name (valid values: "signal", "image", "macro"))

stop_macro(*number_or_title: int | str | None = None*) → None

Stop macro.

Parameters
number_or_title – Macro number, or macro title. Defaults to None (current macro).

Raises
ValueError – if macro not found

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters
state – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters

state – Show titles state

Proxy context manager

The proxy context manager is a convenient way to handle proxy creation and destruction. It is used as follows:

```
with proxy_context("local") as proxy:
    proxy.add_signal(...)
```

The proxy type can be "local" or "remote". For remote proxy, the port can be specified as "remote:port".

Note: The proxy context manager allows to use the proxy in various contexts (Python script, Jupyter notebook, etc.). It also allows to switch seamlessly between local and remote proxy, keeping the same code inside the context.

datalab.control.proxy.**proxy_context**(*what: str*) → Generator[*LocalProxy* | *RemoteProxy*, None, None]

Context manager handling DL proxy creation and destruction.

Parameters

what – proxy type ("local" or "remote") For remote proxy, the port can be specified as "remote:port"

Yields

proxy

LocalProxy if what == "local" RemoteProxy if what == "remote" or "remote:port"

Example

with proxy_context("local") as proxy:

proxy.add_signal(...)

Calling processor methods using proxy objects

All the proxy objects provide access to the DataLab computing methods exposed by the processor classes:

- `datalab.gui.processor.signal.SignalProcessor`
- `datalab.gui.processor.image.ImageProcessor`

To run a computation feature associated to a processor, you can use the `calc()` method of the proxy object:

```
# Call a method without parameter
proxy.calc("average")

# Call a method with parameters
p = sigima.params.MovingAverageParam.create(n=30)
proxy.calc("moving_average", p)
```

GUI

The *datalab.gui* package contains functionnalities related to the graphical user interface (GUI) of the DataLab project. Those features are mostly specific to DataLab and are not intended to be used independently.

The purpose of this section of the documentation is to provide an overview of the DataLab GUI architecture and to describe the main features of the modules contained in this package. It is not intended to provide a detailed description of the GUI features, but rather to provide a starting point for the reader who wants to understand the DataLab internal architecture.

DataLab's main window is composed of several parts, each of them being handled by a specific module of this package:

- **Signal and image panels**: those panels are used to display signals and images and to provide a set of tools to manipulate them. Each data panel relies on a set of modules to handle the GUI features (*datalab.gui.actionhandler* and *datalab.gui.objectview*), the data model (`datalab.gui.objectmodel`), the data visualization (*datalab.gui.plothandler*), and the data processing (*datalab.gui.processor*).
- **Macro panel**: this panel is used to display and run macros. It relies on the `datalab.gui.macroeditor` module to handle the macro edition and execution.
- **Specialized widgets**: those widgets are used to handle specific features such as ROI edition (*datalab.gui.roieditor*), Intensity profile edition (`datalab.gui.profiledialog`), etc.

Submodule	Purpose
datalab.gui.main	DataLab main window and application
datalab.gui.panel	Signal, image and macro panels
datalab.gui.actionhandler	Application actions (menus, toolbars, context menu)
datalab.gui.objectview	Widgets to display object (signal/image) trees
datalab.gui.plothandler	*PlotPy* plot items for representing signals and images
datalab.gui.roieditor	ROI editor
datalab.gui.processor	Processor
datalab.gui.docks	Dock widgets
datalab.gui.h5io	HDF5 input/output

Main window

The *datalab.gui.main* module provides the main window of the DataLab project.

class datalab.gui.main.**DLMainWindow**(*console=None*, *hide_on_close=False*)

DataLab main window

Parameters

- **console** – enable internal console
- **hide_on_close** – True to hide window on close

static get_instance(*console=None*, *hide_on_close=False*)

Return singleton instance

static xmlrpc_server_started(*port*)

XML-RPC server has started, writing comm port in configuration file

get_group_titles_with_object_info() → tuple[list[str], list[list[str]], list[list[str]]]

Return groups titles and lists of inner objects uuids and titles.

Returns
Groups titles, lists of inner objects uuids and titles

`get_object_titles`(*panel: Literal['signal', 'image', 'macro'] | None = None*) → list[str]

Get object (signal/image) list for current panel. Objects are sorted by group number and object index in group.

Parameters
panel – panel name. If None, current data panel is used (i.e. signal or image panel).

Returns
List of object titles

Raises
ValueError – if panel is unknown

`get_object`(*nb_id_title: int | str | None = None, panel: Literal['signal', 'image'] | None = None*) → SignalObj | ImageObj

Get object (signal/image) from index.

Parameters

- **nb_id_title** – Object number, or object id, or object title. Defaults to None (current object).
- **panel** – Panel name. Defaults to None (current panel).

Returns
Object

Raises

- **KeyError** – if object not found
- **TypeError** – if index_id_title type is invalid

`find_object_by_uuid`(*uuid: str*) → SignalObj | ImageObj | ObjectGroup | None

Find an object by UUID, searching across all panels.

This method searches for an object in both signal and image panels, making it suitable for cross-panel operations (e.g., radial profile that takes an ImageObj and produces a SignalObj).

Difference from get_object(): - get_object() requires specifying a panel and accepts number/id/title - find_object_by_uuid() searches all panels automatically using only UUID

Parameters
uuid – UUID of the object to find

Returns
The object if found in any panel, None otherwise

`get_object_uuids`(*panel: Literal['signal', 'image'] | None = None, group: int | str | None = None*) → list[str]

Get object (signal/image) uuid list for current panel. Objects are sorted by group number and object index in group.

Parameters

- **panel** – panel name. If None, current panel is used.
- **group** – Group number, or group id, or group title. Defaults to None (all groups).

Returns
List of object uuids

Raises
`ValueError` – if panel is unknown

`get_sel_object_uuids`(*include_groups: bool = False*) → list[str]

Return selected objects uuids.

Parameters
`include_groups` – If True, also return objects from selected groups.

Returns
List of selected objects uuids.

`add_group`(*title: str, panel: Literal['signal', 'image'] | None = None, select: bool = False*) → None

Add group to DataLab.

Parameters

- `title` – Group title
- `panel` – Panel name. Defaults to None.
- `select` – Select the group after creation. Defaults to False.

`select_objects`(*selection: list[int | str], panel: Literal['signal', 'image'] | None = None*) → None

Select objects in current panel.

Parameters

- `selection` – List of object numbers (1 to N) or uuids to select
- `panel` – panel name. If None, current panel is used. Defaults to None.

`select_groups`(*selection: list[int | str] | None = None, panel: Literal['signal', 'image'] | None = None*) → None

Select groups in current panel.

Parameters

- `selection` – List of group numbers (1 to N), or list of group uuids, or None to select all groups. Defaults to None.
- `panel` – panel name. If None, current panel is used. Defaults to None.

`delete_metadata`(*refresh_plot: bool = True, keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- `refresh_plot` – Refresh plot. Defaults to True.
- `keep_roi` – Keep ROI. Defaults to False.

`get_object_shapes`(*nb_id_title: int | str | None = None, panel: Literal['signal', 'image'] | None = None*) → list

Get plot item shapes associated to object (signal/image).

Parameters

- `nb_id_title` – Object number, or object id, or object title. Defaults to None (current object).
- `panel` – Panel name. Defaults to None (current panel).

Returns
List of plot item shapes

add_annotations_from_items(*items: list, refresh_plot: bool = True, panel: Literal['signal', 'image'] | None = None*) → None

Add object annotations (annotation plot items).

Parameters

- **items** – annotation plot items
- **refresh_plot** – refresh plot. Defaults to True.
- **panel** – panel name. If None, current panel is used.

add_label_with_title(*title: str | None = None, panel: Literal['signal', 'image'] | None = None*) → None

Add a label with object title on the associated plot

Parameters

- **title** – Label title. Defaults to None. If None, the title is the object title.
- **panel** – panel name. If None, current panel is used.

run_macro(*number_or_title: int | str | None = None*) → None

Run macro.

Parameters

number – Number of the macro (starting at 1). Defaults to None (run current macro, or does nothing if there is no macro).

stop_macro(*number_or_title: int | str | None = None*) → None

Stop macro.

Parameters

number – Number of the macro (starting at 1). Defaults to None (stop current macro, or does nothing if there is no macro).

import_macro_from_file(*filename: str*) → None

Import macro from file

Parameters

filename – Filename.

property panels: tuple[*AbstractPanel*, ...]

Return the tuple of implemented panels (signal, image)

Returns

Tuple of panels

confirm_memory_state() → bool

Check memory warning state and eventually show a warning dialog

Returns

True if memory state is ok

check_stable_release() → None

Check if this is a stable release

check_for_previous_crash() → None

Check for previous crash

check_for_v020_plugins() → None

Check for v0.20 plugins and warn user if any are found

execute_post_show_actions() → None

Execute post-show actions

take_screenshot(*name: str*) → None

Take main window screenshot

take_menu_screenshots() → None

Take menu screenshots

setup(*console: bool = False*) → None

Setup main window

Parameters
console – True to setup console

set_process_isolation_enabled(*state: bool*) → None

Enable/disable process isolation

Parameters
state – True to enable process isolation

get_current_panel() → str

Return current panel name

Returns
"signal", "image", "macro")

Return type
Panel name (valid values

set_current_panel(*panel: Literal['signal', 'image', 'macro'] | BaseDataPanel*) → None

Switch to panel.

Parameters
panel – panel name or panel instance

Raises
ValueError – unknown panel

calc(*name: str, param: gds.DataSet | None = None*) → None

Call computation feature name

Note: This calls either the processor's compute_<name> method (if it exists), or the processor's <name> computation feature (if it is registered, using the run_feature method). It looks for the function in all panels, starting with the current one.

Parameters

- **name** – Compute function name
- **param** – Compute function parameter. Defaults to None.

Raises
ValueError – unknown function

has_objects() → bool

Return True if sig/ima panels have any object

set_modified(*state: bool = True*) → None

Set mainwindow modified state

repopulate_panel_trees() → None

Repopulate all panel trees

toggle_show_titles(*state: bool*) → None

Toggle show annotations option

Parameters
state – state

handle_autorefresh_action(*state: bool*) → None

Handle auto-refresh action from UI (with confirmation dialog)

Parameters
state – desired state

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh option

Parameters
state – state

toggle_show_first_only(*state: bool*) → None

Toggle show first only option

Parameters
state – state

reset_all() → None

Reset all application data

save_to_h5_file(*filename=None*) → None

Save to a DataLab HDF5 file

Parameters
filename – HDF5 filename. If None, a file dialog is opened.

Raises
IOError – if filename is invalid or file cannot be saved.

open_h5_files(*h5files: list[str] | None = None, import_all: bool | None = None, reset_all: bool | None = None*) → None

Open a DataLab HDF5 file or import from any other HDF5 file.

Parameters

- **h5files** – HDF5 filenames (optionally with dataset name, separated by ":")
- **import_all** – Import all datasets from HDF5 files
- **reset_all** – Reset all application data before importing

browse_h5_files(*filenames: list[str], reset_all: bool*) → None

Browse HDF5 files

Parameters

- **filenames** – HDF5 filenames
- **reset_all** – Reset all application data before importing

import_h5_file(*filename: str, reset_all: bool | None = None*) → None

Import HDF5 file into DataLab

Parameters

- **filename** – HDF5 filename (optionally with dataset name,
- **"** (*separated by*) – “)
- **reset_all** – Delete all DataLab signals/images before importing data

add_object(*obj: SignalObj | ImageObj, group_id: str = '', set_current=True*) → None

Add object - signal or image

Parameters

- **obj** – object to add (signal or image)
- **group_id** – group ID (optional)
- **set_current** – True to set the object as current object

load_from_files(*filenames: list[str]*) → None

Open objects from files in current panel (signals/images)

Parameters
filenames – list of filenames

load_from_directory(*path: str*) → None

Open objects from directory in current panel (signals/images).

Parameters
path – directory path

get_version() → str

Return DataLab public version.

Returns
DataLab version

close_application() → None

Close DataLab application

raise_window() → None

Raise DataLab window

add_signal(*title: str, xdata: ndarray, ydata: ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add signal data to DataLab.

Parameters

- **title** – Signal title
- **xdata** – X data
- **ydata** – Y data
- **xunit** – X unit. Defaults to “”
- **yunit** – Y unit. Defaults to “”
- **xlabel** – X label. Defaults to “”
- **ylabel** – Y label. Defaults to “”

- **group_id** – group id in which to add the signal. Defaults to ""
- **set_current** – if True, set the added signal as current

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(*title: str, data: ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True*) → bool

Add image data to DataLab.

Parameters

- **title** – Image title
- **data** – Image data
- **xunit** – X unit. Defaults to ""
- **yunit** – Y unit. Defaults to ""
- **zunit** – Z unit. Defaults to ""
- **xlabel** – X label. Defaults to ""
- **ylabel** – Y label. Defaults to ""
- **zlabel** – Z label. Defaults to ""
- **group_id** – group id in which to add the image. Defaults to ""
- **set_current** – if True, set the added image as current

Returns
True if image was added successfully, False otherwise

Raises
ValueError – Invalid data dtype

play_demo() → None

Play demo

show_tour() → None

Show tour

static test_segfault_error() → None

Generate errors (both fault and traceback)

show() → None

Reimplement QMainWindow method

close_properly() → bool

Close properly

Returns
True if closed properly, False otherwise

closeEvent(*event: QCloseEvent*) → None

Reimplement QMainWindow method

Panel

The `datalab.gui.panel` package provides the **panel objects** for signals and images.

Three types of panels are available:

- `datalab.gui.panel.signal.SignalPanel`: Signal panel
- `datalab.gui.panel.image.ImagePanel`: Image panel
- `datalab.gui.panel.macro.MacroPanel`: Macro panel

Signal and Image Panels are called **Data Panels** and are used to display and handle signals and images in the main window of DataLab.

Data Panels rely on the `datalab.gui.panel.base.ObjectProp` class (managing the object properties) and a set of modules to handle the GUI features:

- `datalab.gui.actionhandler`: Application actions (menus, toolbars, context menu)
- `datalab.gui.objectview`: Widgets to display object (signal/image) trees
- `datalab.gui.plothandler`: *PlotPy* items for representing signals and images
- `datalab.gui.processor`: Processor (computation)
- `datalab.gui.panel.roieditor`: ROI editor

The Macro Panel is used to display and run macros. It relies on the `datalab.gui.macroeditor` module to handle the macro edition and execution.

Base features

`datalab.gui.panel.base.`**`is_plot_item_serializable`**(*item: Any*) → bool

Return True if plot item is serializable

`datalab.gui.panel.base.`**`is_hdf5_file`**(*filename: str*, *check_content: bool = False*) → bool

Return True if filename has an HDF5 extension or is an HDF5 file.

Parameters

- **filename** – Path to the file to check
- **check_content** – If True, also attempts to open the file to verify it's a valid HDF5 file. If False, only checks the file extension.

Returns

True if the file is (likely) an HDF5 file, False otherwise.

class `datalab.gui.panel.base.`**`ProcessingReport`**(*success: bool*, *obj_uuid: str | None = None*, *message: str | None = None*)

Report of processing operation

Parameters

- **success** – True if processing succeeded
- **obj_uuid** – UUID of the processed object
- **message** – Optional message (error or info)

class datalab.gui.panel.base.**ObjectProp**(*panel:* BaseDataPanel, *objclass: SignalObj | ImageObj*)

Object handling panel properties

Parameters

- **panel** – parent data panel
- **objclass** – class of the object handled by the panel (SignalObj or ImageObj)

display_analysis_parameters(*obj: SignalObj | ImageObj*) → bool

Set analysis parameter label.

Parameters
obj – Signal or Image object

Returns
True if analysis parameters were found and displayed, False otherwise.

display_processing_history(*obj: SignalObj | ImageObj*) → bool

Display processing history.

Parameters
obj – Signal or Image object

Returns
True if processing history was found and displayed, False otherwise.

update_properties_from(*obj: SignalObj | ImageObj | None = None*) → None

Update properties panel (properties, creation, processing) from object.

Parameters
obj – Signal or Image object

get_changed_properties() → dict[str, Any]

Get dictionary of properties that have changed from original values.

Returns
Dictionary mapping property names to their new values, containing only the properties that were modified by the user.

update_original_values() → None

Update the stored original values to the current dataset values.

This should be called after applying changes to reset the baseline for detecting future changes.

setup_creation_tab(*obj: SignalObj | ImageObj, set_current: bool = False*) → bool

Setup the Creation tab with parameter editor for interactive object creation.

Parameters

- **obj** – Signal or Image object
- **set_current** – If True, set the Creation tab as current after creation

Returns
True if Creation tab was set up, False otherwise

apply_creation_parameters() → None

Apply creation parameters: recreate object with updated parameters.

setup_processing_tab(*obj: SignalObj | ImageObj*, *reset_params: bool = True*, *set_current: bool = False*) → bool

Setup the Processing tab with parameter editor for re-processing.

Parameters

- **obj** – Signal or Image object
- **reset_params** – If True, call update_from_obj() to reset parameters from source object. If False, use parameters as stored in metadata.
- **set_current** – If True, set the Processing tab as current after creation

Returns
True if Processing tab was set up, False otherwise

apply_processing_parameters(*obj: SignalObj | ImageObj | None = None*, *interactive: bool = True*) → *ProcessingReport*

Apply processing parameters: re-run processing with updated parameters.

Parameters

- **obj** – Signal or Image object to reprocess. If None, uses the current object.
- **interactive** – If True, show progress and error messages in the UI.

Returns
ProcessingReport with success status, object UUID, and optional message.

class datalab.gui.panel.base.**AbstractPanelMeta**(*name*, *bases*, *namespace*, ***kwargs*)

Mixed metaclass to avoid conflicts

class datalab.gui.panel.base.**AbstractPanel**(*parent*)

Object defining DataLab panel interface, based on a vertical QSplitter widget

A panel handle an object list (objects are signals, images, macros…). Each object must implement `datalab.gui.ObjItf` interface

get_serializable_name(*obj: ObjItf*) → str

Return serializable name of object

serialize_object_to_hdf5(*obj: ObjItf*, *writer: NativeH5Writer*) → None

Serialize object to HDF5 file

deserialize_object_from_hdf5(*reader: NativeH5Reader*, *name: str*, *reset_all: bool = False*) → ObjItf

Deserialize object from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **name** – Object name in HDF5 file
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate UUIDs (importing objects).

abstract serialize_to_hdf5(*writer: NativeH5Writer*) → None

Serialize whole panel to a HDF5 file

abstract deserialize_from_hdf5(*reader: NativeH5Reader*, *reset_all: bool = False*) → None

Deserialize whole panel from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate UUIDs (importing objects).

abstract create_object() → ObjItf

Create and return object

abstract add_object(*obj: ObjItf*) → None

Add object to panel

abstract remove_all_objects()

Remove all objects

class datalab.gui.panel.base.**PasteMetadataParam**

Paste metadata parameters

keep_roi

Default: True.

Type
guidata.dataset.dataitems.BoolItem

keep_geometry

Default: False.

Type
guidata.dataset.dataitems.BoolItem

keep_tables

Default: False.

Type
guidata.dataset.dataitems.BoolItem

keep_other

Default: True.

Type
guidata.dataset.dataitems.BoolItem

classmethod create(*keep_roi: bool*, *keep_geometry: bool*, *keep_tables: bool*, *keep_other: bool*) → *datalab.gui.panel.base.PasteMetadataParam*

Returns a new instance of `PasteMetadataParam` with the fields set to the given values.

Parameters

- **keep_roi** (*bool*) – Default: True.
- **keep_geometry** (*bool*) – Default: False.
- **keep_tables** (*bool*) – Default: False.
- **keep_other** (*bool*) – Default: True.

Returns
New instance of `PasteMetadataParam`.

class datalab.gui.panel.base.**NonModalInfoDialog**(*parent: QWidget*, *title: str*, *text: str*)

Non-modal information message box with selectable text.

This widget displays an information message in a message dialog box, allowing users to select and copy the text content.

class datalab.gui.panel.base.**SaveToDirectoryGUIParam**

Save to directory parameters

directory

Default: ''.

Type
guidata.dataset.dataitems.DirectoryItem

basename

Basename pattern. Python format string. See description for details. Default: '{title}'.

Type
guidata.dataset.dataitems.StringItem

help

Default: None.

Type
guidata.dataset.dataitems.ButtonItem

extension

Default: None.

Type
guidata.dataset.dataitems.ChoiceItem

overwrite

Overwrite existing files. Default: False.

Type
guidata.dataset.dataitems.BoolItem

preview

String, regexp: `^(?!invalid).\*`. Default: None.

Type
guidata.dataset.dataitems.TextItem

classmethod create(*directory: str*, *basename: str*, *help: type*, *extension: Any*, *overwrite: bool*, *preview: str*) → *datalab.gui.panel.base.SaveToDirectoryGUIParam*

Returns a new instance of *SaveToDirectoryGUIParam* with the fields set to the given values.

Parameters

- **directory** (*str*) – Default: ''.
- **basename** (*str*) – Basename pattern. Python format string. See description for details. Default: '{title}'.
- **help** (*type*) – Default: None.
- **extension** (*Any*) – Default: None.
- **overwrite** (*bool*) – Overwrite existing files. Default: False.
- **preview** (*str*) – String, regexp: `^(?!invalid).\*`. Default: None.

Returns
New instance of *SaveToDirectoryGUIParam*.

on_button_click(*_item: ButtonItem*, *_value: None*, *parent: QWidget*) → None

Help button callback.

get_extension_choices(*_item=None*, *_value=None*)

Return list of available extensions for choice item.

build_filenames(*objs: list[TypeObj] | None = None*) → list[str]

Build filenames according to current parameters.

generate_filepath_obj_pairs(*objs: list[TypeObj]*) → Generator[tuple[str, TypeObj], None, None]

Iterate over (filepath, object) pairs to be saved.

update_preview(*_item=None*, *_value=None*) → None

Update preview.

class datalab.gui.panel.base.**AddMetadataParam**

Add metadata parameters

metadata_key

The key name for the metadata item. String, not empty, regexp: `^[a-za-z_][a-za-z0-9_]\*$`. Default: 'custom_key'.

Type
guidata.dataset.dataitems.StringItem

value_pattern

Python format string. See description for details. Default: '{index}'.

Type
guidata.dataset.dataitems.StringItem

help

Default: None.

Type
guidata.dataset.dataitems.ButtonItem

conversion

Default: 'string'.

Type
guidata.dataset.dataitems.ChoiceItem

preview

String, regexp: `^(?!invalid).\*`. Default: ''.

Type
guidata.dataset.dataitems.TextItem

classmethod create(*metadata_key: str*, *value_pattern: str*, *help: type*, *conversion: Any*, *preview: str*) → *datalab.gui.panel.base.AddMetadataParam*

Returns a new instance of `AddMetadataParam` with the fields set to the given values.

Parameters

- **metadata_key** (*str*) – The key name for the metadata item. String, not empty, regexp: `^[a-za-z_][a-za-z0-9_]\*$`. Default: 'custom_key'.
- **value_pattern** (*str*) – Python format string. See description for details. Default: '{index}'.

- **help** (*type*) – Default: None.
- **conversion** (*Any*) – Default: 'string'.
- **preview** (*str*) – String, regexp: `^(?!invalid).\*`. Default: ''.

Returns
New instance of *AddMetadataParam*.

on_help_button_click(*_item: ButtonItem*, *_value: None*, *parent: QWidget*) → None

Help button callback.

get_conversion_choices(*_item=None*, *_value=None*)

Return list of available conversion choices.

build_values(*objs: list[TypeObj] | None = None*) → list[str | float | int | bool]

Build values according to current parameters.

Raises
ValueError – If a value cannot be converted to the target type.

update_preview(*_item=None*, *_value=None*) → None

Update preview.

class datalab.gui.panel.base.**BaseDataPanel**(*parent: QWidget*)

Object handling the item list, the selected item properties and plot

abstract static get_roi_class() → Type[TypeROI]

Return ROI class

abstract static get_roieditor_class() → Type[TypeROIEditor]

Return ROI editor class

closeEvent(*event*)

Reimplement QMainWindow method

plot_item_parameters_changed(*item: CurveItem | MaskedXYImageItem | LabelItem*) → None

Plot items changed: update metadata of all objects from plot items

plot_item_moved(*item: LabelItem*, *x0: float*, *y0: float*, *x1: float*, *y1: float*) → None

Plot item moved: update metadata of all objects from plot items

Parameters

- **item** – Plot item
- **x0** – new x0 coordinate
- **y0** – new y0 coordinate
- **x1** – new x1 coordinate
- **y1** – new y1 coordinate

serialize_object_to_hdf5(*obj: TypeObj*, *writer: NativeH5Writer*) → None

Serialize object to HDF5 file

serialize_to_hdf5(*writer: NativeH5Writer*) → None

Serialize whole panel to a HDF5 file

deserialize_from_hdf5(*reader: NativeH5Reader*, *reset_all: bool = False*) → None

Deserialize whole panel from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate UUIDs (importing objects).

create_object() → TypeObj

Create object (signal or image)

Returns
SignalObj or ImageObj object

add_object(*obj: TypeObj*, *group_id: str | None = None*, *set_current: bool = True*) → None

Add object

Parameters

- **obj** – SignalObj or ImageObj object
- **group_id** – group id to which the object belongs. If None or empty string, the object is added to the current group.
- **set_current** – if True, set the added object as current

remove_all_objects() → None

Remove all objects

setup_panel() → None

Setup panel

refresh_plot(*what: str*, *update_items: bool = True*, *force: bool = False*, *only_visible: bool = True*, *only_existing: bool = False*) → None

Refresh plot.

Parameters

- **what** – string describing the objects to refresh. Valid values are "selected" (refresh the selected objects), "all" (refresh all objects), "existing" (refresh existing plot items), or an object uuid.
- **update_items** – if True, update the items. If False, only show the items (do not update them). Defaults to True.
- **force** – if True, force refresh even if auto refresh is disabled. Defaults to False.
- **only_visible** – if True, only refresh visible items. Defaults to True. Visible items are the ones that are not hidden by other items or the items except the first one if the option "Show first only" is enabled. This is useful for images, where the last image is the one that is shown. If False, all items are refreshed.
- **only_existing** – if True, only refresh existing items. Defaults to False. Existing items are the ones that have already been created and are associated to the object uuid. If False, create new items for the objects that do not have an item yet.

Raises
ValueError – if *what* is not a valid value

manual_refresh() → None

Manual refresh

get_category_actions(*category:* ActionCategory) → list[QAction]

Return actions for category

get_context_menu() → QMenu

Update and return context menu

add_group(*title: str*, *select: bool = False*) → ObjectGroup

Add group

Parameters

- **title** – group title
- **select** – if True, select the group in the tree view. Defaults to False.

Returns

Created group object

duplicate_object() → None

Duplication signal/image object

copy_metadata() → None

Copy object metadata

paste_metadata(*param:* PasteMetadataParam | *None = None*) → None

Paste metadata to selected object(s)

add_metadata(*param:* AddMetadataParam | *None = None*) → None

Add metadata item to selected object(s)

Parameters

param – Add metadata parameters

copy_roi() → None

Copy regions of interest

paste_roi() → None

Paste regions of interest

remove_object(*force: bool = False*) → None

Remove signal/image object

Parameters

force – if True, remove object without confirmation. Defaults to False.

delete_all_objects() → None

Confirm before removing all objects

delete_metadata(*refresh_plot: bool = True*, *keep_roi: bool | None = None*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep regions of interest, if any. Defaults to None (ask user).

add_annotations_from_items(*items: list*, *refresh_plot: bool = True*) → None

Add object annotations (annotation plot items).

Parameters

- **items** – annotation plot items
- **refresh_plot** – refresh plot. Defaults to True.

update_metadata_view_settings() → None

Update metadata view settings

copy_titles_to_clipboard() → None

Copy object titles to clipboard (for reproducibility)

new_group() → None

Create a new group

rename_selected_object_or_group(*new_name: str | None = None*) → None

Rename selected object or group

Parameters

new_name – new name (default: None, i.e. ask user)

abstract get_newparam_from_current(*newparam: NewSignalParam | NewImageParam | None = None*) → NewSignalParam | NewImageParam | None

Get new object parameters from the current object.

Parameters

newparam – new object parameters. If None, create a new one.

Returns

New object parameters

abstract new_object(*param: NewSignalParam | NewImageParam | None = None*, *edit: bool = False*, *add_to_panel: bool = True*) → TypeObj | None

Create a new object (signal/image).

Parameters

- **param** – new object parameters
- **edit** – Open a dialog box to edit parameters (default: False). When False, the object is created with default parameters and creation parameters are stored in metadata for interactive editing.
- **add_to_panel** – Add object to panel (default: True)

Returns

New object

set_current_object_title(*title: str*) → None

Set current object title

load_from_directory(*directory: str | None = None*) → list[TypeObj]

Open objects from directory (signals or images, depending on the panel), add them to DataLab and return them. If the directory is not specified, ask the user to select a directory.

Parameters

directory – directory name

Returns
list of new objects

`load_from_files`(*filenames: list[str] | None = None, create_group: bool = False, add_objects: bool = True, ignore_errors: bool = False*) → list[TypeObj]

Open objects from file (signals/images), add them to DataLab and return them.

Parameters

- **filenames** – File names
- **create_group** – if True, create a new group if more than one object is loaded for a single file. Defaults to False: all objects are added to the current group.
- **add_objects** – if True, add objects to the panel. Defaults to True.
- **ignore_errors** – if True, ignore errors when loading files. Defaults to False.

Returns
list of new objects

`save_to_files`(*filenames: list[str] | str | None = None*) → None

Save selected objects to files (signal/image).

Parameters
filenames – File names

`save_to_directory`(*param: SaveToDirectoryParam | None = None*) → None

Save signals or images to directory using a filename pattern.

Opens a dialog to select the output directory, the basename pattern and the extension.

Parameters
param – parameters.

`handle_dropped_files`(*filenames: list[str] | None = None*) → None

Handle dropped files

Parameters
filenames – File names

Returns
None

`exec_import_wizard`() → None

Execute import wizard

`import_metadata_from_file`(*filename: str | None = None*) → None

Import metadata from file (JSON).

Parameters
filename – File name

`export_metadata_from_file`(*filename: str | None = None*) → None

Export metadata to file (JSON).

Parameters
filename – File name

`copy_annotations`() → None

Copy annotations from selected object

paste_annotations() → None

Paste annotations to selected object(s)

import_annotations_from_file(*filename: str | None = None*) → None

Import annotations from file (JSON).

Parameters
filename – File name

export_annotations_from_file(*filename: str | None = None*) → None

Export annotations to file (JSON).

Parameters
filename – File name

delete_annotations() → None

Delete all annotations from selected object(s)

import_roi_from_file(*filename: str | None = None*) → None

Import regions of interest from file (JSON).

Parameters
filename – File name

export_roi_to_file(*filename: str | None = None*) → None

Export regions of interest to file (JSON).

Parameters
filename – File name

selection_changed(*update_items: bool = False*) → None

Object selection changed: update object properties, refresh plot and update object view.

Parameters
update_items – Update plot items (default: False)

properties_changed() → None

The properties 'Apply' button was clicked: update object properties, refresh plot and update object view.

recompute_processing() → None

Recompute/rerun selected objects or group with stored processing parameters.

This method handles both single objects and groups. For each object, it checks if it has 1-to-1 processing parameters that can be recomputed. Objects without recomputable parameters are skipped.

select_source_objects() → None

Select source objects associated with the selected object's processing.

This method retrieves the source object UUIDs from the selected object's processing parameters and selects them in the object view.

add_plot_items_to_dialog(*dlg: PlotDialog, oids: list[str]*) → None

Add plot items to dialog

Parameters

- **dlg** – Dialog
- **oids** – Object IDs

open_separate_view(*oids: list[str] | None = None, edit_annotations: bool = False*) → PlotDialog | None

Open separate view for visualizing selected objects

Parameters

- **oids** – Object IDs (default: None)
- **edit_annotations** – Edit annotations (default: False)

Returns
Instance of PlotDialog

view_images_side_by_side(*oids: list[str] | None = None*) → None

View selected images side-by-side in a grid layout

Parameters
oids – Object IDs (default: None, uses selected objects)

get_dialog_size() → tuple[int, int]

Get dialog size (minimum and maximum)

create_new_dialog(*edit: bool = False, toolbar: bool = True, title: str | None = None, name: str | None = None, options: dict[str, Any] | None = None*) → PlotDialog | None

Create new pop-up signal/image plot dialog.

Parameters

- **edit** – Edit mode
- **toolbar** – Show toolbar
- **title** – Dialog title
- **name** – Dialog object name
- **options** – Plot options

Returns
Plot dialog instance

get_roi_editor_output(*mode: Literal['apply', 'extract', 'define'] = 'apply'*) → tuple[TypeROI, bool] | None

Get ROI data (array) from specific dialog box.

Parameters
mode – Mode of operation, either "apply" (define ROI, then apply it to selected objects), "extract" (define ROI, then extract data from it), or "define" (define ROI without applying or extracting).

Returns
A tuple containing the ROI object and a boolean indicating whether the dialog was accepted or not.

get_objects_with_dialog(*title: str, comment: str = '', nb_objects: int = 1, parent: QW.QWidget | None = None*) → TypeObj | None

Get object with dialog box.

Parameters

- **title** – Dialog title
- **comment** – Optional dialog comment
- **nb_objects** – Number of objects to select

- **parent** – Parent widget

Returns
Object(s) (signal(s) or image(s), or None if dialog was canceled)

show_results() → None

Show results

toggle_result_label_visibility(*state: bool*) → None

Toggle the visibility of the merged result label on the plot.

Parameters
state – True to show the label, False to hide it

plot_results(*kind: str | None = None, xaxis: str | None = None, yaxis: str | None = None*) → None

Plot results

Parameters

- **kind** – Plot kind. Either "one_curve_per_object" or "one_curve_per_title". If None, show dialog to get parameters.
- **xaxis** – X axis column name. If None, show dialog to get parameters.
- **yaxis** – Y axis column name. If None, show dialog to get parameters.

delete_results() → None

Delete results

add_label_with_title(*title: str | None = None, ignore_msg: bool = True*) → None

Add a label with object title on the associated plot

Parameters

- **title** – Label title. Defaults to None. If None, the title is the object title.
- **ignore_msg** – If True, do not show the information message. Defaults to True. If False, show a message box to inform the user that the label has been added as an annotation, and that it can be edited or removed using the annotation editing window.

Signal panel

class datalab.gui.panel.signal.**SignalPanel**(*parent: QW.QWidget, dockableplotwidget: DockablePlotWidget, panel_toolbar: QW.QToolBar*)

Object handling the item list, the selected item properties and plot, specialized for Signal objects

PARAMCLASS

Signal object

title

Signal title. Default: 'Untitled'.
Type
guidata.dataset.dataitems.StringItem

xydata

Default: None.
Type
guidata.dataset.dataitems.FloatArrayItem

metadata

Default: {}.

Type

guidata.dataset.dataitems.DictItem

annotations

Default: ''.

Type

guidata.dataset.dataitems.StringItem

xlabel

Title. Default: ''.

Type

guidata.dataset.dataitems.StringItem

xunit

Default: ''.

Type

guidata.dataset.dataitems.StringItem

ylabel

Title. Default: ''.

Type

guidata.dataset.dataitems.StringItem

yunit

Default: ''.

Type

guidata.dataset.dataitems.StringItem

autoscale

Default: True.

Type

guidata.dataset.dataitems.BoolItem

xscalelog

Default: False.

Type

guidata.dataset.dataitems.BoolItem

xscalemin

Lower bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

xscalemax

Upper bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

yscalelog

Default: False.

Type

guidata.dataset.dataitems.BoolItem

yscalemin

Lower bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

yscalemax

Upper bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

alias of `SignalObj`

classmethod `PARAMCLASS.create`(*title: str, xydata: numpy.ndarray, metadata: dict, annotations: str, xlabel: str, xunit: str, ylabel: str, yunit: str, autoscale: bool, xscalelog: bool, xscalemin: float, xscalemax: float, yscalelog: bool, yscalemin: float, yscalemax: float*) → sigima.objects.signal.object.SignalObj

Returns a new instance of `SignalObj` with the fields set to the given values.

Parameters

- **title** (*str*) – Signal title. Default: 'Untitled'.
- **xydata** (*numpy.ndarray*) – Default: None.
- **metadata** (*dict*) – Default: {}.
- **annotations** (*str*) – Default: ''.
- **xlabel** (*str*) – Title. Default: ''.
- **xunit** (*str*) – Default: ''.
- **ylabel** (*str*) – Title. Default: ''.
- **yunit** (*str*) – Default: ''.
- **autoscale** (*bool*) – Default: True.
- **xscalelog** (*bool*) – Default: False.
- **xscalemin** (*float*) – Lower bound. Default: None.
- **xscalemax** (*float*) – Upper bound. Default: None.
- **yscalelog** (*bool*) – Default: False.
- **yscalemin** (*float*) – Lower bound. Default: None.
- **yscalemax** (*float*) – Upper bound. Default: None.

Returns

New instance of `SignalObj`.

IO_REGISTRY

alias of `SignalIORegistry`

static get_roi_class() → Type[SignalROI]

Return ROI class

static get_roieditor_class() → Type[*SignalROIEditor*]

Return ROI editor class

get_newparam_from_current(*newparam: NewSignalParam | None = None, title: str | None = None*) → NewSignalParam | None

Get new object parameters from the current object.

Parameters

- **newparam** (*guidata.dataset.DataSet*) – new object parameters. If None, create a new one.
- **title** – new object title. If None, use the current object title, or the default title.

Returns
New object parameters

new_object(*param: NewSignalParam | None = None, edit: bool = False, add_to_panel: bool = True*) → SignalObj | None

Create a new object (signal).

Parameters

- **param** (*guidata.dataset.DataSet*) – new object parameters
- **edit** (*bool*) – Open a dialog box to edit parameters (default: False). When False, the object is created with default parameters and creation parameters are stored in metadata for interactive editing.
- **add_to_panel** (*bool*) – Add the new object to the panel (default: True)

Returns
New object

toggle_anti_aliasing(*state: bool*) → None

Toggle anti-aliasing on/off

Parameters
state – state of the anti-aliasing

reset_curve_styles() → None

Reset curve styles

Image panel

class datalab.gui.panel.image.**ImagePanel**(*parent: QW.QWidget, dockableplotwidget: DockablePlotWidget, panel_toolbar: QW.QToolBar*)

Object handling the item list, the selected item properties and plot, specialized for Image objects

PARAMCLASS

Image object

data

Default: None.
Type
guidata.dataset.dataitems.FloatArrayItem

metadata

Default: {}.
Type
guidata.dataset.dataitems.DictItem

annotations

Default: ''.

Type

guidata.dataset.dataitems.StringItem

is_uniform_coords

Default: True.

Type

guidata.dataset.dataitems.BoolItem

x0

X_0. Default: 0.0.

Type

guidata.dataset.dataitems.FloatItem

y0

Y_0. Default: 0.0.

Type

guidata.dataset.dataitems.FloatItem

dx

x. Default: 1.0.

Type

guidata.dataset.dataitems.FloatItem

dy

y. Default: 1.0.

Type

guidata.dataset.dataitems.FloatItem

xmin

X_{MIN}. Default: None.

Type

guidata.dataset.dataitems.FloatItem

xmax

X_{MAX}. Default: None.

Type

guidata.dataset.dataitems.FloatItem

ymin

Y_{MIN}. Default: None.

Type

guidata.dataset.dataitems.FloatItem

ymax

Y_{MAX}. Default: None.

Type

guidata.dataset.dataitems.FloatItem

xcoords

X coordinates. Default: array([], dtype=float64).

Type

guidata.dataset.dataitems.FloatArrayItem

ycoords

Y coordinates. Default: array([], dtype=float64).

Type

guidata.dataset.dataitems.FloatArrayItem

title

Image title. Default: 'Untitled'.

Type

guidata.dataset.dataitems.StringItem

xlabel

Title. Default: ''.

Type

guidata.dataset.dataitems.StringItem

xunit

Default: ''.

Type

guidata.dataset.dataitems.StringItem

ylabel

Title. Default: ''.

Type

guidata.dataset.dataitems.StringItem

yunit

Default: ''.

Type

guidata.dataset.dataitems.StringItem

zlabel

Title. Default: ''.

Type

guidata.dataset.dataitems.StringItem

zunit

Default: ''.

Type

guidata.dataset.dataitems.StringItem

autoscale

Default: True.

Type

guidata.dataset.dataitems.BoolItem

xscalelog

Default: False.

Type

guidata.dataset.dataitems.BoolItem

xscalemin

Lower bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

xscalemax

Upper bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

yscalelog

Default: False.

Type

guidata.dataset.dataitems.BoolItem

yscalemin

Lower bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

yscalemax

Upper bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

zscalemin

Lower bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

zscalemax

Upper bound. Default: None.

Type

guidata.dataset.dataitems.FloatItem

alias of `ImageObj`

classmethod PARAMCLASS.**create**(*data: numpy.ndarray, metadata: dict, annotations: str, is_uniform_coords: bool, x0: float, y0: float, dx: float, dy: float, xmin: float, xmax: float, ymin: float, ymax: float, xcoords: numpy.ndarray, ycoords: numpy.ndarray, title: str, xlabel: str, xunit: str, ylabel: str, yunit: str, zlabel: str, zunit: str, autoscale: bool, xscalelog: bool, xscalemin: float, xscalemax: float, yscalelog: bool, yscalemin: float, yscalemax: float, zscalemin: float, zscalemax: float*) → sigima.objects.image.object.ImageObj

Returns a new instance of `ImageObj` with the fields set to the given values.

Parameters

- **data** (*numpy.ndarray*) – Default: None.
- **metadata** (*dict*) – Default: {}.
- **annotations** (*str*) – Default: ''.
- **is_uniform_coords** (*bool*) – Default: True.
- **x0** (*float*) – X_0. Default: 0.0.
- **y0** (*float*) – Y_0. Default: 0.0.
- **dx** (*float*) – x. Default: 1.0.
- **dy** (*float*) – y. Default: 1.0.
- **xmin** (*float*) – X_{MIN}. Default: None.

- **xmax** (*float*) – X_{MAX}. Default: None.
- **ymin** (*float*) – Y_{MIN}. Default: None.
- **ymax** (*float*) – Y_{MAX}. Default: None.
- **xcoords** (*numpy.ndarray*) – X coordinates. Default: array([], dtype=float64).
- **ycoords** (*numpy.ndarray*) – Y coordinates. Default: array([], dtype=float64).
- **title** (*str*) – Image title. Default: 'Untitled'.
- **xlabel** (*str*) – Title. Default: ''.
- **xunit** (*str*) – Default: ''.
- **ylabel** (*str*) – Title. Default: ''.
- **yunit** (*str*) – Default: ''.
- **zlabel** (*str*) – Title. Default: ''.
- **zunit** (*str*) – Default: ''.
- **autoscale** (*bool*) – Default: True.
- **xscalelog** (*bool*) – Default: False.
- **xscalemin** (*float*) – Lower bound. Default: None.
- **xscalemax** (*float*) – Upper bound. Default: None.
- **yscalelog** (*bool*) – Default: False.
- **yscalemin** (*float*) – Lower bound. Default: None.
- **yscalemax** (*float*) – Upper bound. Default: None.
- **zscalemin** (*float*) – Lower bound. Default: None.
- **zscalemax** (*float*) – Upper bound. Default: None.

Returns
New instance of `ImageObj`.

IO_REGISTRY
alias of `ImageIORegistry`

static get_roi_class() → Type[ImageROI]
Return ROI class

static get_roieditor_class() → Type[*ImageROIEditor*]
Return ROI editor class

plot_lut_changed(*plot: BasePlot*) → None
The LUT of the plot has changed: updating image objects accordingly

Parameters
plot – Plot object

get_newparam_from_current(*newparam: NewImageParam | None = None, title: str | None = None*) → NewImageParam | None
Get new object parameters from the current object.

Parameters

- **newparam** (*guidata.dataset.DataSet*) – new object parameters. If None, create a new one.
- **title** – new object title. If None, use the current object title, or the default title.

Returns
New object parameters

new_object(*param: NewImageParam | None = None, edit: bool = False, add_to_panel: bool = True*) → ImageObj | None

Create a new object (image).

Parameters

- **param** (*guidata.dataset.DataSet*) – new object parameters
- **edit** (*bool*) – Open a dialog box to edit parameters (default: False). When False, the object is created with default parameters and creation parameters are stored in metadata for interactive editing.
- **add_to_panel** (*bool*) – Add the object to the panel (default: True)

Returns
New object

toggle_show_contrast(*state: bool*) → None

Toggle show contrast option

Macro panel

class datalab.gui.panel.macro.**MacroTabs**(*parent=None*)

Macro tabwidget

Parameters
parent (*QWidget*) – Parent widget

clear() → None

Override Qt method

contextMenuEvent(*event*)

Override Qt method

add_tab(*macro: Macro*) → int

Add tab

Parameters
macro – Macro object

Returns
Number of the tab (starting at 1)

Return type
int

remove_tab(*number: int*) → None

Remove tab

Parameters
number – Number of the tab (starting at 1)

get_widget(*number: int*) → CodeEditor

Return macro editor widget at number

Parameters
number – Number of the tab (starting at 1)

Returns
Macro editor widget

set_current_number(*number: int*) → None

Set current tab number

Parameters
number – Number of the tab (starting at 1)

get_current_number() → int

Return current tab number

Returns
Number of the tab (starting at 1)

Return type
int

set_tab_title(*number: int*, *name: str*) → None

Set tab title

Parameters

- **number** – Number of the tab (starting at 1)
- **name** – Macro name

class datalab.gui.panel.macro.**MacroPanel**(*parent: QMainWindow*)

Macro Panel widget

Parameters
parent (*QWidget*) – Parent widget

update_color_mode() → None

Update color mode according to the current theme

get_serializable_name(*obj: Macro*) → str

Return serializable name of object

serialize_to_hdf5(*writer: NativeH5Writer*) → None

Serialize whole panel to a HDF5 file

Parameters
writer – HDF5 writer

deserialize_from_hdf5(*reader: NativeH5Reader*, *reset_all: bool = False*) → None

Deserialize whole panel from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate UUIDs (importing objects).

create_object() → Macro

Create object.

Returns
Macro object

add_object(*obj: Macro*) → None

Add object.

Parameters
obj – Macro object

remove_all_objects() → None

Remove all objects

setup_actions() → None

Setup macro menu actions

get_macro(*number_or_title: int | str | None = None*) → Macro | None

Return macro at number (if number is None, return current macro)

Parameters
number – Number of the macro (starting at 1) or title of the macro. Defaults to None (current macro).

Returns
Macro object or None (if not found)

get_number_from_title(*title: str*) → int | None

Return macro number from title

Parameters
title – Title of the macro

Returns
Number of the macro (starting at 1) or None (if not found)

get_number_from_macro(*macro: Macro*) → int | None

Return macro number from macro object

Parameters
macro – Macro object

Returns
Number of the macro (starting at 1) or None (if not found)

get_macro_titles() → list[str]

Return list of macro titles

macro_contents_changed() → None

One of the macro contents has changed

run_macro(*number_or_title: int | str | None = None*) → None

Run current macro

Parameters
number – Number of the macro (starting at 1). Defaults to None (run current macro, or does nothing if there is no macro).

stop_macro(*number_or_title: int | str | None = None*) → None

Stop current macro

Parameters

number – Number of the macro (starting at 1). Defaults to None (run current macro, or does nothing if there is no macro).

macro_state_changed(*orig_macro: Macro, state: bool*) → None

Macro state has changed (True: started, False: stopped)

Parameters

- **orig_macro** – Macro object
- **state** – State of the macro

add_macro() → Macro

Add macro, optionally with name

Returns

Macro object

macro_name_changed(*name: str*) → None

Macro name has been changed

Parameters

name – New name of the macro

rename_macro(*number: int | None = None, title: str | None = None*) → None

Rename macro

Parameters

- **number** – Number of the macro (starting at 1). Defaults to None.
- **title** – Title of the macro. Defaults to None.

export_macro_to_file(*number_or_title: int | str | None = None, filename: str | None = None*) → None

Export macro to file

Parameters

- **number_or_title** – Number of the macro (starting at 1) or title of the macro. Defaults to None.
- **filename** – Filename. Defaults to None.

Raises

ValueError – If title is not found

import_macro_from_file(*filename: str | None = None*) → int

Import macro from file

Parameters

filename – Filename. Defaults to None.

Returns

Number of the macro (starting at 1)

remove_macro(*number_or_title: int | str | None = None*) → None

Remove macro

Parameters
number_or_title – Number of the macro (starting at 1) or title of the macro. Defaults to None.

Action handler

The *datalab.gui.actionhandler* module handles all application actions (menus, toolbars, context menu). These actions point to DataLab panels, processors, objecthandler, …

Utility classes

class datalab.gui.actionhandler.**SelectCond**

Signal or image select conditions

static always(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Always true

static exactly_one(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Exactly one signal or image is selected

static exactly_one_group(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Exactly one group is selected

static exactly_one_group_or_one_object(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Exactly one group or one signal or image is selected

static at_least_one_group_or_one_object(*sel_groups: list[ObjectGroup], sel_objects: list[SignalObj | ImageObj]*) → bool

At least one group or one signal or image is selected

static at_least_one(*sel_groups: list[ObjectGroup], sel_objects: list[SignalObj | ImageObj]*) → bool

At least one signal or image is selected

static at_least_two(*sel_groups: list[ObjectGroup], sel_objects: list[SignalObj | ImageObj]*) → bool

At least two signals or images are selected

static with_roi(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

At least one signal or image has a ROI

static exactly_one_with_roi(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Exactly one signal or image has a ROI

static exactly_one_with_annotations(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Exactly one signal or image has annotations

static with_annotations(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

At least one signal or image has annotations

static with_results(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

At least one signal or image has results

class datalab.gui.actionhandler.**ActionCategory**(*value*)

Action categories

Handler classes

class datalab.gui.actionhandler.**SignalActionHandler**(*panel: SignalPanel | ImagePanel, panel_toolbar: QW.QToolBar, view_toolbar: QW.QToolBar*)

Object handling signal panel GUI interactions: actions, menus, …

create_first_actions()

Create actions that are added to the menus in the first place

create_last_actions()

Create actions that are added to the menus in the end

action_for(*function_or_name: Callable | str, position: int | None = None, separator: bool = False, context_menu_pos: int | None = None, context_menu_sep: bool = False, toolbar_pos: int | None = None, toolbar_sep: bool = False, toolbar_category: ActionCategory | None = None*) → QW.QAction

Create action for a feature.

Parameters

- **function_or_name** – function or name of the feature
- **position** – add action to menu at this position. Defaults to None.
- **separator** – add separator before action in menu
- **context_menu_pos** – add action to context menu at this position.
- **context_menu_pos** – add action to context menu at this position. Defaults to None.
- **context_menu_sep** – add separator before action in context menu (or after if context_menu_pos is positive). Defaults to False.
- **toolbar_pos** – add action to toolbar at this position. Defaults to None.
- **toolbar_sep** – add separator before action in toolbar (or after if toolbar_pos is positive). Defaults to False.
- **toolbar_category** – toolbar category. Defaults to None. If toolbar_pos is not None, this specifies the category of the toolbar. If None, defaults to ActionCategory.VIEW_TOOLBAR if the current category is ActionCategory.VIEW, else to ActionCategory.PANEL_TOOLBAR.

Returns

New action

add_action(*action: QW.QAction, select_condition: Callable | None = None*) → None

Add action to list of actions.

Parameters

- **action** – action to add

- **select_condition** – condition to enable action. Defaults to None. If None, action is enabled if at least one object is selected.

add_to_action_list(*action: QW.QAction*, *category:* ActionCategory | *None* = *None*, *pos: int* | *None* = *None*, *sep: bool* = *False*) → None

Add action to list of actions.

Parameters

- **action** – action to add
- **category** – action category. Defaults to None. If None, action is added to the current category.
- **pos** – add action to menu at this position. Defaults to None. If None, action is added at the end of the list.
- **sep** – add separator before action in menu (or after if pos is positive). Defaults to False.

create_all_actions()

Create all actions

has_annotations_in_clipboard(*selected_groups: list[ObjectGroup]*, *selected_objects: list[SignalObj* | *ImageObj]*) → bool

Check if annotations clipboard is not empty

has_metadata_in_clipboard(*selected_groups: list[ObjectGroup]*, *selected_objects: list[SignalObj* | *ImageObj]*) → bool

Check if metadata clipboard is not empty

new_action(*title: str*, *position: int* | *None* = *None*, *separator: bool* = *False*, *triggered: Callable* | *None* = *None*, *toggled: Callable* | *None* = *None*, *shortcut: QW.QShortcut* | *None* = *None*, *icon_name: str* | *None* = *None*, *tip: str* | *None* = *None*, *select_condition: Callable* | *str* | *None* = *None*, *context_menu_pos: int* | *None* = *None*, *context_menu_sep: bool* = *False*, *toolbar_pos: int* | *None* = *None*, *toolbar_sep: bool* = *False*, *toolbar_category:* ActionCategory | *None* = *None*) → QW.QAction

Create new action and add it to list of actions.

Parameters

- **title** – action title
- **position** – add action to menu at this position. Defaults to None.
- **separator** – add separator before action in menu (or after if pos is positive). Defaults to False.
- **triggered** – triggered callback. Defaults to None.
- **toggled** – toggled callback. Defaults to None.
- **shortcut** – shortcut. Defaults to None.
- **icon_name** – icon name. Defaults to None.
- **tip** – tooltip. Defaults to None.
- **select_condition** – selection condition. Defaults to None. If str, must be the name of a method of SelectCond, i.e. one of "always", "exactly_one", "exactly_one_group", "at_least_one_group_or_one_object", "at_least_one", "at_least_two", "with_roi".
- **context_menu_pos** – add action to context menu at this position. Defaults to None.

- **context_menu_sep** – add separator before action in context menu (or after if context_menu_pos is positive). Defaults to False.
- **toolbar_pos** – add action to toolbar at this position. Defaults to None.
- **toolbar_sep** – add separator before action in toolbar (or after if toolbar_pos is positive). Defaults to False.
- **toolbar_category** – toolbar category. Defaults to None. If toolbar_pos is not None, this specifies the category of the toolbar. If None, defaults to ActionCategory.VIEW_TOOLBAR if the current category is ActionCategory.VIEW, else to ActionCategory.PANEL_TOOLBAR.

Returns
New action

new_category(*category: ActionCategory*) → Generator[None, None, None]

Context manager for creating a new menu.

Parameters
category – Action category

Yields
None

new_menu(*title: str*, *icon_name: str | None = None*, *store_ref: str | None = None*) → Generator[None, None, None]

Context manager for creating a new menu.

Parameters

- **title** – Menu title
- **icon_name** – Menu icon name. Defaults to None.
- **store_ref** – Optional attribute name to store menu reference. Defaults to None.

Yields
None

property object_suffix: str

Object suffix (e.g. "sig" for signal, "ima" for image)

populate_results_delete_submenu() → None

Populate the Results Delete submenu dynamically based on current selection

populate_roi_remove_submenu() → None

Populate the ROI Remove submenu dynamically based on current selection

selected_objects_changed(*selected_groups: list[ObjectGroup]*, *selected_objects: list[SignalObj | ImageObj]*) → None

Update actions based on selected objects.

Parameters

- **selected_groups** – selected groups
- **selected_objects** – selected objects

class datalab.gui.actionhandler.**ImageActionHandler**(*panel: SignalPanel | ImagePanel*, *panel_toolbar: QW.QToolBar*, *view_toolbar: QW.QToolBar*)

Object handling image panel GUI interactions: actions, menus, …

create_first_actions()

Create actions that are added to the menus in the first place

create_last_actions()

Create actions that are added to the menus in the end

action_for(*function_or_name: Callable | str, position: int | None = None, separator: bool = False, context_menu_pos: int | None = None, context_menu_sep: bool = False, toolbar_pos: int | None = None, toolbar_sep: bool = False, toolbar_category: ActionCategory | None = None*) → QW.QAction

Create action for a feature.

Parameters

- **function_or_name** – function or name of the feature
- **position** – add action to menu at this position. Defaults to None.
- **separator** – add separator before action in menu
- **context_menu_pos** – add action to context menu at this position.
- **context_menu_pos** – add action to context menu at this position. Defaults to None.
- **context_menu_sep** – add separator before action in context menu (or after if context_menu_pos is positive). Defaults to False.
- **toolbar_pos** – add action to toolbar at this position. Defaults to None.
- **toolbar_sep** – add separator before action in toolbar (or after if toolbar_pos is positive). Defaults to False.
- **toolbar_category** – toolbar category. Defaults to None. If toolbar_pos is not None, this specifies the category of the toolbar. If None, defaults to ActionCategory.VIEW_TOOLBAR if the current category is ActionCategory.VIEW, else to ActionCategory.PANEL_TOOLBAR.

Returns

New action

add_action(*action: QW.QAction, select_condition: Callable | None = None*) → None

Add action to list of actions.

Parameters

- **action** – action to add
- **select_condition** – condition to enable action. Defaults to None. If None, action is enabled if at least one object is selected.

add_to_action_list(*action: QW.QAction, category: ActionCategory | None = None, pos: int | None = None, sep: bool = False*) → None

Add action to list of actions.

Parameters

- **action** – action to add
- **category** – action category. Defaults to None. If None, action is added to the current category.
- **pos** – add action to menu at this position. Defaults to None. If None, action is added at the end of the list.

- **sep** – add separator before action in menu (or after if pos is positive). Defaults to False.

create_all_actions()

Create all actions

has_annotations_in_clipboard(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Check if annotations clipboard is not empty

has_metadata_in_clipboard(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → bool

Check if metadata clipboard is not empty

new_action(*title: str, position: int | None = None, separator: bool = False, triggered: Callable | None = None, toggled: Callable | None = None, shortcut: QW.QShortcut | None = None, icon_name: str | None = None, tip: str | None = None, select_condition: Callable | str | None = None, context_menu_pos: int | None = None, context_menu_sep: bool = False, toolbar_pos: int | None = None, toolbar_sep: bool = False, toolbar_category: ActionCategory | None = None*) → QW.QAction

Create new action and add it to list of actions.

Parameters

- **title** – action title
- **position** – add action to menu at this position. Defaults to None.
- **separator** – add separator before action in menu (or after if pos is positive). Defaults to False.
- **triggered** – triggered callback. Defaults to None.
- **toggled** – toggled callback. Defaults to None.
- **shortcut** – shortcut. Defaults to None.
- **icon_name** – icon name. Defaults to None.
- **tip** – tooltip. Defaults to None.
- **select_condition** – selection condition. Defaults to None. If str, must be the name of a method of SelectCond, i.e. one of "always", "exactly_one", "exactly_one_group", "at_least_one_group_or_one_object", "at_least_one", "at_least_two", "with_roi".
- **context_menu_pos** – add action to context menu at this position. Defaults to None.
- **context_menu_sep** – add separator before action in context menu (or after if context_menu_pos is positive). Defaults to False.
- **toolbar_pos** – add action to toolbar at this position. Defaults to None.
- **toolbar_sep** – add separator before action in toolbar (or after if toolbar_pos is positive). Defaults to False.
- **toolbar_category** – toolbar category. Defaults to None. If toolbar_pos is not None, this specifies the category of the toolbar. If None, defaults to ActionCategory.VIEW_TOOLBAR if the current category is ActionCategory.VIEW, else to ActionCategory.PANEL_TOOLBAR.

Returns

New action

new_category(*category: ActionCategory*) → Generator[None, None, None]

Context manager for creating a new menu.

Parameters
category – Action category

Yields
None

new_menu(*title: str, icon_name: str | None = None, store_ref: str | None = None*) → Generator[None, None, None]

Context manager for creating a new menu.

Parameters

- **title** – Menu title
- **icon_name** – Menu icon name. Defaults to None.
- **store_ref** – Optional attribute name to store menu reference. Defaults to None.

Yields
None

property object_suffix: str

Object suffix (e.g. "sig" for signal, "ima" for image)

populate_results_delete_submenu() → None

Populate the Results Delete submenu dynamically based on current selection

populate_roi_remove_submenu() → None

Populate the ROI Remove submenu dynamically based on current selection

selected_objects_changed(*selected_groups: list[ObjectGroup], selected_objects: list[SignalObj | ImageObj]*) → None

Update actions based on selected objects.

Parameters

- **selected_groups** – selected groups
- **selected_objects** – selected objects

Object view

The `datalab.gui.objectview` module provides widgets to display object (signal/image) trees.

Note: This module provides tree widgets to display signals, images and groups. It is important to note that, by design, the user can only select either individual signals/images or groups, but not both at the same time. This is an important design choice, as it allows to simplify the user experience, and to avoid potential confusion between the two types of selection.

Simple object tree

class datalab.gui.objectview.**SimpleObjectTree**(*parent: QW.QWidget*, *objmodel: ObjectModel*)

Base object handling panel list widget, object (sig/ima) lists

initialize_from(*sobjlist:* SimpleObjectTree) → None

Init from another SimpleObjectList, without making copies of objects

iter_items(*item: QW.QTreeWidgetItem* | *None* = *None*) → Iterator[QW.QTreeWidgetItem]

Recursively iterate over all items

get_item_from_id(*item_id*) → QTreeWidgetItem

Return QTreeWidgetItem from id (stored in item's data)

get_current_item_id(*object_only: bool* = *False*) → str | None

Return current item id

set_current_item_id(*uuid: str*, *extend: bool* = *False*) → None

Set current item by id

get_current_group_id() → str

Return current group ID

get_sel_group_items() → list[QTreeWidgetItem]

Return selected group items

get_sel_group_uuids() → list[str]

Return selected group uuids

get_sel_object_items() → list[QTreeWidgetItem]

Return selected object items

get_sel_object_uuids(*include_groups: bool* = *False*) → list[str]

Return selected objects uuids.

Parameters

include_groups – If True, also return objects from selected groups.

Returns

List of selected objects uuids.

get_sel_objects(*include_groups: bool* = *False*) → list[SignalObj | ImageObj]

Return selected objects.

If include_groups is True, also return objects from selected groups.

get_sel_groups() → list[ObjectGroup]

Return selected groups

populate_tree() → None

Populate tree with objects

update_tree() → None

Update tree

add_group_item(*group: ObjectGroup*) → None

Add group item

add_object_item(*obj: SignalObj | ImageObj*, *group_id: str*, *set_current: bool = True*) → None

Add item

update_item(*uuid: str*) → None

Update item

remove_item(*oid: str*, *refresh: bool = True*) → None

Remove item

item_double_clicked(*item: QTreeWidgetItem*) → None

Item was double-clicked: open a pop-up plot dialog

contextMenuEvent(*event: QContextMenuEvent*) → None

Override Qt method

Get object dialog

class datalab.gui.objectview.**GetObjectsDialog**(*parent: QW.QWidget*, *panel: BaseDataPanel*, *title: str*, *comment: str = ''*, *nb_objects: int = 1*, *minimum_size: tuple[int, int] | None = None*)

Dialog box showing groups and objects (signals or images) to select one, or more.

Parameters

- **parent** – parent widget
- **panel** – data panel
- **title** – dialog title
- **comment** – optional dialog comment
- **nb_objects** – number of objects to select (default: 1)
- **minimum_size** – minimum size (width, height)

get_selected_objects() → list[SignalObj | ImageObj]

Return selected objects

Object view

class datalab.gui.objectview.**ObjectView**(*parent: BaseDataPanel*, *objmodel: ObjectModel*)

Object handling panel list widget, object (sig/ima) lists

paintEvent(*event*)

Reimplement Qt method

dragEnterEvent(*event: QDragEnterEvent*) → None

Reimplement Qt method

dragLeaveEvent(*event: QDragLeaveEvent*) → None

Reimplement Qt method

dragMoveEvent(*event: QDragMoveEvent*) → None

Reimplement Qt method

get_all_group_uuids() → list[str]

Return all group uuids, in a list ordered by group position in the tree

get_all_object_uuids() → dict[str, list[str]]

Return all object uuids, in a dictionary that maps group uuids to the list of object uuids in each group, in the correct order

dropEvent(*event: QDropEvent*) → None

Reimplement Qt method

get_current_object() → SignalObj | ImageObj | None

Return current object

set_current_object(*obj: SignalObj | ImageObj*) → None

Set current object

item_selection_changed() → None

Refreshing the selection of objects and groups, emitting the SIG_SELECTION_CHANGED signal which triggers the update of the object properties panel, the plot items and the actions of the toolbar and menu bar.

This method is called when the user selects or deselects items in the tree. It is also called when the user clicks on an item that was already selected.

This method emits the SIG_SELECTION_CHANGED signal.

select_objects(*selection: list[SignalObj | ImageObj | int | str]*) → None

Select multiple objects

Parameters

selection – list of objects, object numbers (1 to N) or object uuids

select_groups(*groups: list[ObjectGroup | int | str] | None = None*) → None

Select multiple groups

Parameters

groups – list of groups, group numbers (1 to N), group names or None (select all groups). Defaults to None.

move_up()

Move selected objects/groups up

move_down()

Move selected objects/groups down

Plot handler

The `datalab.gui.plothandler` module provides plot handlers for signal and image panels, that is, classes handling *PlotPy* plot items for representing signals and images.

Signal plot handler

class datalab.gui.plothandler.**SignalPlotHandler**(*panel:* BaseDataPanel, *plotwidget: PlotWidget*)

Object handling signal plot items, plot dialogs, plot options

toggle_anti_aliasing(*state: bool*) → None

Toggle anti-aliasing

Parameters
state – if True, enable anti-aliasing

get_plot_options() → PlotOptions

Return standard signal/image plot options

refresh_plot(*what: str*, *update_items: bool = True*, *force: bool = False*, *only_visible: bool = True*, *only_existing: bool = False*) → None

Refresh plot and configure datetime axis if needed.

This override adds automatic datetime axis configuration when at least one of the displayed signals has a datetime X-axis.

Parameters

- **what** – string describing the objects to refresh
- **update_items** – if True, update the items
- **force** – if True, force refresh even if auto refresh is disabled
- **only_visible** – if True, only refresh visible items
- **only_existing** – if True, only refresh existing items

add_shapes(*oid: str*, *do_autoscale: bool = False*) → None

Add geometric shape items associated to computed results and annotations, for the object with the given uuid

cleanup_dataview() → None

Clean up data view

clear() → None

Clear plot items

get(*key: str*, *default: TypePlotItem | None = None*) → TypePlotItem | None

Return item associated to object uuid. If the key is not found, default is returned if given, otherwise None is returned.

get_existing_oids() → list[str]

Get existing object uuids.

Returns
List of object uuids that have a plot item associated to them.

get_obj_from_item(*item: TypePlotItem*) → TypeObj | None

Return object associated to plot item

Parameters
item – plot item

Returns
Object associated to plot item

reduce_shown_oids(*oids: list[str]*) → list[str]

Reduce the number of shown objects to visible items only. The base implementation is to show only the first selected item if the option "Show first only" is enabled.

Parameters
oids – list of object uuids

Returns
Reduced list of object uuids

refresh_all_shape_items() → None

Refresh all geometric shapes to apply new style parameters.

This method is called when shape/marker visualization settings are changed in the Settings dialog. It removes and recreates all shape items to apply the new parameters.

remove_all_shape_items() → None

Remove all geometric shapes associated to result items

remove_item(*oid: str*) → None

Remove plot item associated to object uuid

set_auto_refresh(*auto_refresh: bool*) → None

Set auto refresh mode.

Parameters
auto_refresh – if True, refresh plot items automatically

set_show_first_only(*show_first_only: bool*) → None

Set show first only mode.

Parameters
show_first_only – if True, show only the first selected item

toggle_result_label_visibility(*show: bool*) → None

Toggle the visibility of all merged result labels on the plot.

Parameters
show – True to show the labels, False to hide them

update_resultproperty_from_plot_item(*item: LabelItem*) → None

Update result properties from merged label plot item.

When the merged results label is moved, update the stored position in the merged result adapter.

Parameters
item – Merged results label item

Image plot handler

class datalab.gui.plothandler.**ImagePlotHandler**(*panel: BaseDataPanel, plotwidget: PlotWidget*)

Object handling image plot items, plot dialogs, plot options

reduce_shown_oids(*oids: list[str]*) → list[str]

Reduce the number of shown objects to visible items only. The base implementation is to show only the first selected item if the option "Show first only" is enabled.

Parameters
oids – list of object uuids

Returns
Reduced list of object uuids

refresh_plot(*what: str*, *update_items: bool = True*, *force: bool = False*, *only_visible: bool = True*, *only_existing: bool = False*) → None

Refresh plot.

Parameters

- **what** – string describing the objects to refresh. Valid values are "selected" (refresh the selected objects), "all" (refresh all objects), "existing" (refresh existing plot items), or an object uuid.
- **update_items** – if True, update the items. If False, only show the items (do not update them). Defaults to True.
- **force** – if True, force refresh even if auto refresh is disabled. Defaults to False.
- **only_visible** – if True, only refresh visible items. Defaults to True. Visible items are the ones that are not hidden by other items or the items except the first one if the option "Show first only" is enabled. This is useful for images, where the last image is the one that is shown. If False, all items are refreshed.
- **only_existing** – if True, only refresh existing items. Defaults to False. Existing items are the ones that have already been created and are associated to the object uuid. If False, create new items for the objects that do not have an item yet.

Raises
ValueError – if *what* is not a valid value

cleanup_dataview() → None

Clean up data view

get_plot_options() → PlotOptions

Return standard signal/image plot options

add_shapes(*oid: str*, *do_autoscale: bool = False*) → None

Add geometric shape items associated to computed results and annotations, for the object with the given uuid

clear() → None

Clear plot items

get(*key: str*, *default: TypePlotItem | None = None*) → TypePlotItem | None

Return item associated to object uuid. If the key is not found, default is returned if given, otherwise None is returned.

get_existing_oids() → list[str]

Get existing object uuids.

Returns
List of object uuids that have a plot item associated to them.

get_obj_from_item(*item: TypePlotItem*) → TypeObj | None

Return object associated to plot item

Parameters
item – plot item

Returns
Object associated to plot item

refresh_all_shape_items() → None

Refresh all geometric shapes to apply new style parameters.

This method is called when shape/marker visualization settings are changed in the Settings dialog. It removes and recreates all shape items to apply the new parameters.

remove_all_shape_items() → None

Remove all geometric shapes associated to result items

remove_item(*oid: str*) → None

Remove plot item associated to object uuid

set_auto_refresh(*auto_refresh: bool*) → None

Set auto refresh mode.

Parameters
auto_refresh – if True, refresh plot items automatically

set_show_first_only(*show_first_only: bool*) → None

Set show first only mode.

Parameters
show_first_only – if True, show only the first selected item

toggle_result_label_visibility(*show: bool*) → None

Toggle the visibility of all merged result labels on the plot.

Parameters
show – True to show the labels, False to hide them

update_resultproperty_from_plot_item(*item: LabelItem*) → None

Update result properties from merged label plot item.

When the merged results label is moved, update the stored position in the merged result adapter.

Parameters
item – Merged results label item

ROI editor

The `datalab.gui.roieditor` module provides the ROI editor widgets for signals and images.

Signal ROI editor

class datalab.gui.roieditor.**SignalROIEditor**(*parent: QW.QWidget | None, obj: SignalObj | ImageObj, mode: Literal['apply', 'extract', 'define'] = 'apply', item: TypePlotItem | None = None, options: PlotOptions | dict[str, Any] | None = None, size: tuple[int, int] | None = None*)

Signal ROI Editor

Parameters

- **parent** – Parent plot dialog
- **obj** – Object to edit (`sigima.objects.SignalObj` or `sigima.objects.ImageObj`)

- **mode** – Mode of operation, either "apply" (define ROI, then apply it to selected objects), "extract" (define ROI, then extract data from it), or "define" (define ROI without applying or extracting).
- **item** – Optional plot item to add to the plot (if None, a new item is created from the object)

get_obj_roi_class() → type[SignalROI]

Get object ROI class

add_tools_to_plot_dialog() → None

Add tools to plot dialog

manually_add_roi() → None

Manually add segment ROI

create_coordinate_based_roi_actions() → list[QAction]

Create coordinate-based ROI actions

Image ROI editor

class datalab.gui.roieditor.**ImageROIEditor**(*parent: QW.QWidget | None, obj: SignalObj | ImageObj, mode: Literal['apply', 'extract', 'define'] = 'apply', item: TypePlotItem | None = None, options: PlotOptions | dict[str, Any] | None = None, size: tuple[int, int] | None = None*)

Image ROI Editor

Parameters

- **parent** – Parent plot dialog
- **obj** – Object to edit (`sigima.objects.SignalObj` or `sigima.objects.ImageObj`)
- **mode** – Mode of operation, either "apply" (define ROI, then apply it to selected objects), "extract" (define ROI, then extract data from it), or "define" (define ROI without applying or extracting).
- **item** – Optional plot item to add to the plot (if None, a new item is created from the object)

get_obj_roi_class() → type[ImageROI]

Get object ROI class

add_tools_to_plot_dialog() → None

Add tools to plot dialog

manually_add_roi(*roi_type: Literal['rectangle', 'circle', 'polygon']*) → None

Manually add image ROI

create_coordinate_based_roi_actions() → list[QAction]

Create coordinate-based ROI actions

setup_items() → None

Setup items

Processor

The *datalab.gui.processor* package provides the **processor objects** for signals and images.

Processor objects are the bridge between the computation modules (in `sigima.proc`) and the GUI modules (in *datalab.gui*). They are used to call the computation functions and to update the GUI from inside the data panel objects.

When implementing a processing feature in DataLab, the steps are usually the following:

- Add an action in the *datalab.gui.actionhandler* module to trigger the processing feature from the GUI (e.g. a menu item or a toolbar button).
- Implement the computation function in the `sigima.proc` module (that would eventually call the algorithm from the `sigima.tools` module).
- Implement the processor object method in this package to call the computation function and eventually update the GUI.

The processor objects are organized in submodules according to their purpose:

- `datalab.gui.processor.base`: Common processing features
- `datalab.gui.processor.signal`: Signal processing features
- `datalab.gui.processor.image`: Image processing features

Generic processing types

To support consistent processing workflows, the `BaseProcessor` class defines five generic processing methods, each corresponding to a fundamental input/output pattern. These methods are tightly integrated with the GUI logic: the input objects are taken from the current selection in the active panel (Signal or Image), and the output objects are automatically appended to the same panel.

Descriptions:

- `compute_1_to_1`: Applies an independent transformation to each selected object.
- `compute_1_to_0`: Runs an analysis or measurement producing metadata or scalar data.
- `compute_1_to_n`: Produces multiple output objects from a single input (e.g. ROI extraction).
- `compute_n_to_1`: Aggregates multiple objects into one (e.g. sum, average); supports pairwise mode.
- `compute_2_to_1`: Applies a binary operation with a second operand (object or constant); supports pairwise mode.

Method name	Signature	Multi-selection behavior
`compute_1_to_1`	1 object 1 object	k k
`compute_1_to_0`	1 object no object	k 0
`compute_1_to_n`	1 object n objects	k k·n
`compute_n_to_1`	n objects 1 object	n 1 n n (pairwise mode)
`compute_2_to_1`	1 object + 1 operand 1 object	k + 1 k n + n n (pairwise mode)

These methods are for internal or advanced use (e.g. plugin or macro authors) and will evolve without backward compatibility guarantees.

Future developments (such as a visual pipeline editor) may require generalizing this model to support additional sources and destinations beyond the current panel-based selection/output logic.

Docks

The `datalab.gui.docks` module provides the dockable widgets for the DataLab main window.

Plot widget

class datalab.gui.docks.**DataLabPlotWidget**(*plot_type: PlotType*)

DataLab PlotWidget

This class is a subclass of *plotpy.plot.PlotWidget* that provides a customized widget for DataLab, with a specific set of tools and a customized appearance.

Parameters
plot_type – Plot type

register_tools() → None

Register the plotting tools according to the plot type

Dockable plot widget

class datalab.gui.docks.**DockablePlotWidget**(*parent: QWidget*, *plot_type: PlotType*)

Docked plotting widget

Parameters

- **parent** – Parent widget
- **plot_type** – Plot type

setup_layout() → None

Setup layout

update_toolbar_position() → None

Update toolbar position

setup_plotwidget() → None

Setup plotting widget

update_color_mode() → None

Update plot widget styles according to application color mode

get_plot() → BasePlot

Return plot instance

update_watermark(*plot: BasePlot*) → None

Update watermark visibility

visibility_changed(*enable: bool*) → None

DockWidget visibility has changed

HDF5 I/O

The *datalab.gui.h5io* module provides the HDF5 file open/save into/from DataLab data model/main window.

class datalab.gui.h5io.**H5InputOutput**(*mainwindow:* DLMainWindow)

Object handling HDF5 file open/save into/from DataLab data model/main window

Parameters
mainwindow – Main window

save_file(*filename: str*) → None

Save all signals and images from DataLab model into a HDF5 file

open_file(*filename: str*, *import_all: bool*, *reset_all: bool*) → None

Open HDF5 file

import_files(*filenames: list[str]*, *import_all: bool*, *reset_all: bool*) → None

Import HDF5 files

import_dataset_from_file(*filename: str*, *dsetname: str*) → None

Import dataset from HDF5 file

CHAPTER

THREE

CONTRIBUTING

DataLab is **your** platform. If you want it to be improved, you **can** contribute to the project, whether you are a developer or not.

There are many ways to contribute to DataLab project, depending on how much time you have, your experience with open source projects, and your skills.

3.1 Share your ideas and experiences

Besides the classic bug reports and feature requests, you can share your ideas and experiences for improving DataLab. In particular, we are very interested in your feedback on the documentation and tutorials. Moreover, if you have a use case that you would like to share with the community, please let us know.

Without coding, you can contribute to DataLab project by:

- Reporting a bug
- Suggesting an enhancement
- Suggesting a documentation topic
- Suggesting a tutorial topic

3.2 Share your scientific/technical knowledge

Your technical or scientific knowledge is also very valuable to us, as you may directly contribute to the documentation or tutorials. Or, better, if you want to write a tutorial, we will be happy to help you.

Again without coding, you can contribute to DataLab project by:

- Writing documentation
- Writing a tutorial

3.3 Contribute to new features

Even if you are not a developer, you can contribute to the project by:

- Testing new features
- Writing and submitting new Plugins
- Writing and submitting macro-commands

3.4 Develop new features

If you are a developer, you can contribute to the core of the project by fixing bugs or implementing new features.

See *Contribute code* section for more information.

3.4.1 Contribute code

This page explains how you can contribute code to the project.

See also:

The *Coding guidelines* page presents the coding guidelines that you should follow when contributing to the project.

Fork the project

The first step is to fork the project on GitHub. You can do that by visiting DataLab GitHub project and clicking on the "Fork" button on the top right corner of the page.

Once you have forked the project, you will have a copy of the project in your own GitHub account. Then you can clone the project on your computer and start working on it.

Submit a pull request

Once you have made some changes, you can submit a pull request to the original project. To do that, go to your forked project on GitHub and click on the "Pull request" button on the top right corner of the page.

Then you will have to fill a form to describe your pull request. Once you have submitted the pull request, the project maintainers will review your changes and merge them if they are satisfied.

During the review process, the project maintainers will check that your code follows the coding guidelines and that it does not break the existing tests. If your code does not follow the coding guidelines, you will have to fix it before your pull request can be merged.

3.4.2 Coding guidelines

Generic coding guidelines

We follow the PEP 8 coding style.

In particular, we are especially strict about the following guidelines:

- Limit all lines to a maximum of 79 characters.
- Respect the naming conventions (classes, functions, variables, etc.).
- Use specific exceptions instead of the generic `Exception`.

To enforce these guidelines, the following tools are mandatory:

- ruff for code formatting and static code analysis.
- pylint for static code analysis.

ruff

If you are using Visual Studio Code, the project settings will automatically format your code with *ruff* on save (you may also run the task "Run Ruff" to run *ruff* on the project).

To run *ruff*, run the following command:

```
ruff check
```

pylint

To run *pylint*, run the following command:

```
pylint datalab
```

If you are using Visual Studio Code on Windows, you may run the task "Run Pylint" to run *pylint* on the project.

Note: A *pylint* rating greater than 9/10 is required to merge a pull request.

Specific coding guidelines

In addition to the generic coding guidelines, we have the following specific guidelines:

- Write docstrings for all classes, methods and functions. The docstrings should follow the Google style.
- Add typing annotations for all functions and methods. The annotations should use the future syntax (`from __future__ import annotations`)
- Try to keep the code as simple as possible. If you have to write a complex piece of code, try to split it into several functions or classes.
- Add as many comments as possible. The code should be self-explanatory, but it is always useful to add some comments to explain the general idea of the code, or to explain some tricky parts.
- Do not use `from module import *` statements, even in the `__init__` module of a package.

- Avoid using mixins (multiple inheritance) when possible. It is often possible to use composition instead of inheritance.
- Avoid using `__getattr__` and `__setattr__` methods. They are often used to implement lazy initialization, but this can be done in a more explicit way.

3.4.3 Git Workflow

This document describes the Git workflow used in the DataLab project, based on a `main` branch, a `develop` branch, and feature-specific branches. It also defines how bug fixes are managed.

Note: This workflow is a simplified version of the Gitflow Workflow. It has been adapted to suit the needs of the DataLab project at the current stage of development. In the near future, we may consider adopting a more complex workflow, e.g. by adding release branches.

Branching Model

Main Branches

- `main`: Represents the stable, production-ready version of the project.
- `develop`: Used for ongoing development and integration of new features.

Feature Branches

- `feature/feature_name`: Used for the development of new features.
  - Created from `develop`.
  - Merged back into `develop` once completed.
  - Deleted after merging.

Bug Fix Branches

- `fix/xxx`: Used for general bug fixes that are not urgent.
  - Created from `develop`.
  - Merged back into `develop` once completed.
  - Deleted after merging.
- `hotfix/xxx`: Used for urgent production-critical fixes.
  - Created from `main`.
  - Merged back into `main`.
  - The fix is then cherry-picked into `develop`.
  - Deleted after merging.

Note: Hotfixes (high-priority fixes) will be integrated in the next maintenance release (X.Y.Z -> Z+1), while fixes (low-priority fixes) will be integrated in the next feature release (X.Y -> Y+1).

Documentation Branches

When working on documentation that is not related to source code (e.g. training materials, user guides), branches should be named using the `doc/` prefix.

Examples:

- `doc/training-materials`
- `doc/user-guide`

This naming convention improves clarity by clearly separating documentation efforts from code-related development (features, fixes, etc.).

Workflow for New Features

1. Create a new feature branch from `develop`:

```
git checkout develop
git checkout -b develop/feature_name
```

2. Develop the feature and commit changes.
3. Merge the feature branch back into `develop`:

```
git checkout develop
git merge --no-ff develop/feature_name
```

4. Delete the feature branch:

```
git branch -d develop/feature_name
```

Warning: Do not leave feature branches unmerged for too long. Regularly rebase them on `develop` to minimize conflicts.

Workflow for Regular Bug Fixes

1. Create a bug fix branch from `develop`:

```
git checkout develop
git checkout -b fix/bug_description
```

2. Apply the fix and commit changes.
3. Merge the fix branch back into `develop`:

```
git checkout develop
git merge --no-ff fix/bug_description
```

4. Delete the fix branch:

```
git branch -d fix/bug_description
```

Warning: Do not create a `fix/xxx` branch from a `develop/feature_name` branch. Always branch from `develop` to ensure fixes are correctly propagated.

```
# Incorrect:
git checkout develop/feature_name
git checkout -b fix/wrong_branch
```

```
# Correct:
git checkout develop
git checkout -b fix/correct_branch
```

Workflow for Critical Hotfixes

1. Create a hotfix branch from `main`:

```
git checkout main
git checkout -b hotfix/critical_bug
```

2. Apply the fix and commit changes.
3. Merge the fix back into `main`:

```
git checkout main
git merge --no-ff hotfix/critical_bug
```

4. Cherry-pick the fix into `develop`:

```
git checkout develop
git cherry-pick <commit_hash>
```

5. Delete the hotfix branch:

```
git branch -d hotfix/critical_bug
```

Warning: Do not merge `fix/xxx` or `hotfix/xxx` directly into `main` without following the workflow. Ensure hotfixes are cherry-picked into `develop` to avoid losing fixes in future releases.

Best Practices

- Regularly **rebase feature branches** on `develop` to stay up to date:

```
git checkout develop/feature_name
git rebase develop
```

- Avoid long-lived branches to minimize merge conflicts.
- Ensure bug fixes in `main` are **always cherry-picked** to `develop`.
- Clearly differentiate between `fix/xxx` (non-urgent fixes) and `hotfix/xxx` (critical production fixes).

Takeaway

This workflow ensures a structured yet flexible development process while keeping `main` stable and `develop` always updated with the latest changes.

It also ensures that bug fixes are correctly managed and propagated across branches.

3.4.4 About Dependencies

Dependencies are an important part of any software project, and they can significantly affect the development process. In this section, we will discuss the dependencies used in our project, how to manage them, and best practices for keeping them up to date.

Managing Dependencies

Dependencies in DataLab Project are defined in the *pyproject.toml* file, as officially recommended by the Python packaging authority (see Writing your pyproject.toml).

The *pyproject.toml* file contains many sections, but the most relevant for dependencies are:

- *[project.dependencies]*: This section lists the direct dependencies of the application. These are the packages that the application needs to run.
- *[project.optional-dependencies]*: This section lists optional dependencies that can be installed to enable additional features. These dependencies are not required for the core functionality of the application but can enhance its capabilities.

Among the optional dependencies, we have the following groups:

- *qt*: Qt-based graphical user interface (currently, PyQt5).
- *opencv*: computer vision tasks requiring OpenCV.
- *dev*: development and testing (linters, formatters, etc.).
- *doc*: building the documentation (Sphinx, etc.).
- *test*: running the tests (pytest, etc.).

Deploying Dependencies

Production

In production, we recommend using the package manager that is most suitable for your environment and that you are most comfortable with. All package managers (e.g., *pip*, *conda*) are directly or indirectly using the information from the *pyproject.toml* file to install the dependencies.

See *Installation* for more information on how to install DataLab and its dependencies.

Development

In development, you may also use the requirements text file to make it easier to install the dependencies in a virtual environment or container.

The *requirements.txt* file lists all the dependencies needed for the project, including both direct and optional dependencies. This is the exact list of dependencies that are defined in the *pyproject.toml* file, but formatted for use with *pip* or other package managers that support requirements files.

Note: The requirements file is generated from the *pyproject.toml* file using a tool provided by the *guidata* package.

```
python -m guidata.utils.genreqs txt  # to generate requirements.txt
```

Adding New Dependencies

When adding new dependencies to the project, please follow these rules:

1. If it is a direct dependency, that is a package that the application needs to run, add it to the *[project.dependencies]* section in the *pyproject.toml* file, and specify the version range that is compatible with the application, in order to avoid breaking changes.
2. If it is an optional dependency, that is a package that can be installed to enable additional features, add it to the appropriate section in the *[project.optional-dependencies]* section in the *pyproject.toml* file.

- For the *dev* dependencies, unless it's absolutely necessary, do not specify a version range, as it may limit the ability to use the latest version of the package.
- For the other optional dependencies, specify the version range that is compatible with the application, in order to avoid breaking changes.

Note: In other words, except for the *dev* dependencies, always specify a version range that is compatible with the application.

3.4.5 Setting up Development Environment

Getting started with DataLab development is easy.

Here is what you will need:

1. An integrated development environment (IDE) for Python. We recommend Spyder or Visual Studio Code, but any IDE will do.
2. A Python distribution. We recommend WinPython, on Windows, or Anaconda, on Linux or Mac. But, again, any Python distribution will do.
3. A clean project structure (see below).
4. Test data (see below).
5. Environment variables (see below).
6. Third-party software (see below).

Development Environment

If you are using Spyder, thank you for supporting the scientific open-source Python community!

If you are using Visual Studio Code, that's also an excellent choice (for other reasons). We recommend installing the following extensions:

Extension	Description
gettext	Gettext syntax highlighting
Pylance	Python language server
Python	Python extension
reStructuredText Syntax highlighting	reStructuredText syntax highlighting
Ruff	Extremely fast Python linter and code formatter
Todo Tree	Todo tree
Insert GUID	Insert GUID
XML Tools	XML Tools

Python Environment

DataLab requires the following :

- Python (e.g. WinPython)
- Additional Python packages

Installing all required packages :

```
pip install --upgrade -r requirements.txt
```

See Installation for more details on reference Python and Qt versions.

If you are contributing to DataLab's processing features, you will need to work on both DataLab's main repository and sigima's repository. To install `sigima` in editable mode, use the following command:

```
pip install -e ../sigima  # Adjust the path to your local sigima repository
```

If you are using WinPython, thank you for supporting the scientific open-source Python community!

The following table lists the currently officially used Python distributions:

Python version	Status	WinPython version
3.9	OK	3.9.10.0
3.10	OK	3.10.11.1
3.11	OK	3.11.5.0
3.12	OK	3.12.3.0
3.13	OK	3.13.2.0

We strongly recommend using the `.dot` versions of WinPython which are lightweight and can be customized to your needs (using `pip install -r requirements.txt`).

We also recommend using a dedicated WinPython instance for DataLab.

Test data

DataLab test data are located in different folders, depending on their nature or origin.

Required data for unit tests are located in "datalab\data\tests" (public data).

A second folder %DATALAB_DATA% (optional) may be defined for additional tests which are still under development (or for confidential data).

Specific environment variables

Enable the "debug" mode (no stdin/stdout redirection towards internal console):

```
@REM Mode DEBUG
set DEBUG=1
```

Building PDF documentation requires LaTeX. On Windows, the following environment:

```
@REM LaTeX executable must be in Windows PATH, for mathematical equations rendering
@REM Example with MiKTeX :
set PATH=C:\\Apps\\miktex-portable\\texmfs\\install\\miktex\\bin\\x64;%PATH%
```

Visual Studio Code configuration used in `launch.json` and/or `tasks.json` (examples) :

```
@REM Folder containing additional working test data
set DATALAB_DATA=C:\Dev\Projets\DATALAB_data
```

Visual Studio Code `.env` file:

- This file is used to set environment variables for the application.
- It is used to set the `PYTHONPATH` environment variable to the root of the project.
- This is required to be able to import the project modules from within VS Code.
- To create this file, copy the `.env.template` file to `.env` (and eventually add your own paths).

Windows installer

The Windows installer is built using WiX Toolset V4.0.5. Using the WiX Toolset requires .NET SDK V6.0 minimum.

You may install .NET SDK using `winget`:

```
winget install Microsoft.DotNet.SDK.8
```

Once .NET SDK is installed, the WiX Toolset can be installed and configured using the following commands:

```
dotnet tool install --global wix --version 4.0.5
wix extension add WixToolset.UI.wixext/4.0.5
```

First, you need to generate the installer script from a generic template:

```
python wix/makewxs.py
```

Building the installer is done using the following command:

```
wix build .\wix\DataLab.wxs -ext WixToolset.UI.wixext
```

Third-party Software

The following software may be required for maintaining the project:

Software	Description
gettext	Translations
Git	Version control system
ImageMagick	Image manipulation utilities
Inkscape	Vector graphics editor
MikTeX	LaTeX distribution on Windows

3.4.6 Roadmap

This document outlines the current and future development plans for DataLab:

- It includes information about *funding sources*, and *future evolution*.
- It also provides a summary of *past milestones* to give context to the project's evolution.

Funding

> As an open-source project, DataLab relies on the support of various organizations and grants to fund its development and maintenance. The project's roadmap is shaped by the needs and priorities of its users, as well as the availability of resources. Funded work is prioritized and scheduled accordingly, while community contributions are welcomed and encouraged.

From the project's inception in 2023, the development of DataLab has been funded by the following grants and organizations:

Funding	Description
	CEA - French Alternative Energies and Atomic Energy Commission:• DataLab was initially created for analyzing data from CEA's Laser Megajoule (LMJ) facility• Interfaced with the LMJ control system, DataLab is used to process and visualize signals and images acquired with plasma diagnostics (devices such as cameras, digitizers, spectrometers, etc.)• It is also used for R&D activities around the LMJ facility (e.g., metrology, data analysis, etc.)• CEA is the major investor in DataLab and the main contributor to the project: CEA scientists and engineers are actively involved in the roadmap
	CODRA, a software engineering company and software publisher, has supported DataLab's open-source journey since its inception:• Open-source project management and communication (social media, website, etc.)• Conferences and events: SciPy 2024, PyData Paris 2024, Open Source Experience 2024, etc.• Documentation: tutorials, videos, and more
	NLnet Foundation, as part of the NGI0 Commons Fund backed by the European Commission, funded the redesign of DataLab's core architecture — a major overhaul scheduled for inclusion in the 2.0 release (December 2025):• The goal is to decouple the data model, computation, and I/O from the UI• This will enable DataLab to be used as a library in other software

Future Milestones

The following tasks are long-term goals for DataLab. They are not scheduled for any specific release and may evolve over time based on user needs and project direction.

The following table summarizes the future evolutions and maintenance plans for DataLab that are detailed in the sections below.

Milestone type	Description
Future Evolutions	• Support for data acquisition• Web frontend• Support for time series• Database connectors• Jupyter plugin for interactive data analysis• Spyder plugin for interactive data analysis• Jupyter kernel interface for DataLab
Maintenance	• Transition to gRPC for remote control• Drop Qt5 support and migrate to Qt6
Other Tasks	• Create a DataLab plugin template

Support for Data Acquisition

Adding support for data acquisition would be a major step forward in making DataLab not only a platform for signal and image processing, but also a **versatile tool for real-time experimental workflows**. The idea would be to allow users to acquire data directly from various hardware devices (e.g., cameras, digitizers, spectrometers, etc.) **within DataLab itself**.

Such a feature would enable **seamless integration between acquisition and analysis**, allowing users to process and visualize data immediately after it is captured, without needing to switch tools or export/import files.

While no formal design has been established yet, a viable approach could be to:

- Introduce a **new plugin family dedicated to data acquisition**, following the modular architecture of DataLab;
- Define a **generic API for acquisition plugins**, ensuring flexibility and compatibility across device types;

- **Leverage existing solutions** such as PyMoDAQ, a Python-based data acquisition framework already compatible with a wide range of laboratory instruments.

A potential **collaboration with the PyMoDAQ development team** could be explored, to benefit from their ecosystem and avoid duplicating efforts. This would also help foster interoperability and promote open standards in the Python scientific instrumentation community.

Web Frontend

As DataLab's modular architecture evolves, a natural next step is to provide a **web-based frontend** to complement the existing desktop application.

A web frontend would allow users to:

- **Run DataLab remotely** (e.g. from a server) and access it via a browser;
- Perform processing and visualization tasks without needing a local Python environment;
- Facilitate **collaborative data analysis**, sharing sessions or results with colleagues;
- Integrate with JupyterHub, dashboards, or lab management tools for centralized usage.

This frontend could be built on top of the Sigima library (that was recently created from the externalization of DataLab's processing functionalities), exposing its features through a web interface — possibly leveraging tools like **JupyterLab extensions**, **Panel**, or **Dash**, depending on the chosen stack.

While still exploratory, this direction would increase **accessibility and portability** of DataLab, especially in academic and industrial environments.

> The web frontend shall not replace the desktop application, but rather complement it by providing a different access mode. DataLab's philosophy is to remain a **local application** that does not require a server to run, ensuring data privacy and security. More precisely, the web frontend will empower users to run DataLab on a server and access it remotely - typically on a local network, but it will not be a cloud-based solution or a first step towards a fully cloud-based application which would be against the project's philosophy.

Support for Time Series

DataLab currently focuses on generic signal and image processing, but many use cases — especially in scientific instrumentation and experimental physics — involve **time series data**.

Adding dedicated support for time series would make it easier to:

- Handle signals with associated timestamps or non-uniform sampling;
- Perform time-aware processing and visualization (e.g., event alignment, time-based filtering, resampling);
- Integrate with external systems generating time-indexed data.

This feature is tracked in Issue #27, where potential use cases and design considerations are discussed.

Introducing robust time series handling would broaden DataLab's applicability in domains such as data logging, slow control, and real-time monitoring.

Database Connectors

Currently, DataLab operates primarily on files and in-memory data structures. Adding **connectors to databases** would significantly extend its capabilities, especially for users dealing with large, structured, or historical datasets.

This feature would allow DataLab to:

- **Query and load data** from SQL or NoSQL databases (e.g. PostgreSQL, SQLite, MongoDB, etc.);
- Support metadata-driven workflows, where experiments or datasets are referenced from a database;
- **Store analysis results** or annotations back into a database for traceability and reproducibility;
- Facilitate integration into lab information management systems (LIMS) or enterprise data infrastructures.

The design could include:

- A **plugin-based system** for supporting various database backends;
- A simple configuration interface for connection settings and query management;
- Integration with pandas or SQLAlchemy for flexible data exchange.

This evolution would help bridge the gap between **data acquisition, analysis, and long-term storage**, enabling more robust scientific data workflows.

Jupyter Plugin for Interactive Data Analysis

Although DataLab can already be remotely controlled from a Jupyter notebook — thanks to its existing remote control capabilities (see *Remote controlling*) — the user experience could be greatly enhanced by developing a **dedicated Jupyter plugin**.

This plugin would provide a **tighter integration** between Jupyter and DataLab, offering the following features:

- Use DataLab as a **Jupyter kernel**, enabling direct access to its processing capabilities from within a notebook;
- **Display numerical results from DataLab in Jupyter**, and vice versa — for example, importing results computed in a Jupyter notebook into the DataLab interface;
- Allow **interactive data manipulation**: use DataLab for efficient signal and image operations, and Jupyter for custom or home-made data analysis routines;
- Bridge scripting and GUI workflows, making DataLab more attractive for scientific users familiar with Jupyter environments.

Technically, this plugin could rely on the **Jupyter kernel interface** to expose DataLab's capabilities in an interactive programming context.

Such integration would reinforce DataLab's role in the scientific Python ecosystem and facilitate reproducible, notebook-driven analysis workflows.

Spyder Plugin for Interactive Data Analysis

A plugin for the Spyder IDE would address the **same use cases as the Jupyter plugin**, providing seamless integration between DataLab and Spyder's interactive environment.

This plugin would allow users to:

- Interact with DataLab directly from Spyder;
- Visualize or send data between the DataLab GUI and the Spyder console;
- Use DataLab's processing capabilities in real time while scripting custom analysis workflows in Spyder.

As with the Jupyter integration, this plugin could be implemented by leveraging the **Jupyter kernel interface** (see *Remote controlling*), which Spyder already supports internally.

Such a plugin would enhance DataLab's usability for scientists and engineers who prefer Spyder's integrated development environment for exploratory analysis.

Jupyter Kernel Interface for DataLab

Implementing a native **Jupyter kernel interface** for DataLab would provide a more integrated way to use it from other environments such as **Jupyter notebooks, Spyder, or Visual Studio Code**.

This interface would allow:

- Direct control of DataLab from third-party tools that support Jupyter kernels;
- **Two-way data exchange**: e.g., displaying DataLab results inside a notebook, or visualizing Jupyter-generated data inside the DataLab GUI;
- Tighter integration into scripting workflows and IDEs beyond simple remote control.

However, based on initial exploration, implementing a full Jupyter kernel may be **non-trivial and potentially time-consuming**. Given that remote control already enables communication between DataLab and Jupyter (see *Remote controlling*), the added value of a full kernel integration should be carefully evaluated against its complexity and maintenance cost.

This option remains open for discussion depending on user demand and development resources.

Maintenance Plan

2026: Transition to gRPC for Remote Control

DataLab currently relies on XML-RPC for remote control, which may become a limitation for more advanced or high-performance use cases. If the need arises for a more **efficient, robust, and extensible communication protocol**, a switch to **gRPC** is under consideration.

This improvement is tracked in Issue #18.

2025: Drop Qt5 Support and Migrate to Qt6

With the **end-of-life of Qt5 scheduled for mid-2025**, DataLab will fully migrate to **Qt6**. This transition is expected to be **straightforward**, thanks to:

- The use of the `qtpy` abstraction layer;
- The fact that the `PlotPyStack` library is already compatible with Qt6.

This change will ensure compatibility with future versions of Qt and modern Python environments.

Other Tasks

Create a DataLab Plugin Template

To encourage community contributions and facilitate the development of extensions, a **template for creating DataLab plugins** is planned.

This template would:

- Provide a ready-to-use scaffold with best practices;
- Help new developers quickly understand the plugin architecture;
- Promote consistency and modularity across third-party plugins.

The task is tracked in Issue #26.

Past Milestones

From version 0.9 to 1.0, DataLab has undergone significant development and enhancements. The project has evolved from a simple data analysis tool to a powerful platform for processing and visualizing signals and images.

Those enhancements have been made possible thanks to the support of the following organizations (see *Funding* for more details):

- CEA: French Alternative Energies and Atomic Energy Commission
- CODRA: software engineering company and software publisher
- NLnet: NLnet Foundation, as part of the NGI0 Commons Fund

The following table summarizes the major past milestones of DataLab, including the release dates and a brief description of the features or enhancements introduced in each version.

Mile-stone	Description
1.020	• Interactive object editing: modify creation/processing parameters and recompute objects• New object creation: parametric signal/image generators, Poisson noise, complex-valued objects• Enhanced processing: pulse features, ideal filters, 2D resampling, convolution, projections• ROI & annotations: copy/paste, import/export, grid creation, inverse ROI logic• Visualization: merged result labels, datetime support, performance optimizations• File formats: FT-Lab (.sig, .ima), coordinated text files, improved HDF5 support• Non-uniform coordinates and polynomial calibration• Public API enhancements for programmability
0.202	• ANDOR SIF images: add support for background images• Array editor: new copy all, export, and paste features• Fourier analysis: new zero padding feature for signals and images• ROI editor: ROI selection tool now active by default• Signal processing: X-Y mode, abscissa/ordinate find features, full width at y=…• Public API: add `group_id` and `set_current` arguments to add methods
0.192	• Open all signals or images from a folder (recursively)• ROI editor: add options to create ROIs from coordinates
0.182	• New pairwise operand mode (the operation is done on each pair of signals/images)• New polygonal ROI feature• Support Windows 7 SP1 to Windows 11 with a single installer
0.172	• Introduce ROI support across all processing features• Add arithmetic operations on signals and images• New Ubuntu package for native installation on Linux• New Conda package for all platforms (Windows, Linux, MacOS)
0.162	• New validation process for signal and image features• Add support for binary images (unlocking new processing features)
0.152	• New MSI installer for the stand-alone version on Windows• Add support for large text/CSV files (> 1 GB)• Add auto downsampling feature
0.142	• HDF5 browser: multiple file support, detailed info on groups and attributes• New Debian package for native installation on Linux
0.122	• Add a tour and demo feature• Add tutorials to the documentation• Add a Text file import assistant
0.112	• Add features for reordering signals and images (e.g. drag-and-drop)• Add 1D convolution, interpolation, resampling and detrending features
0.102 12	• Develop a very simple DataLab plugin to demonstrate the plugin system• Allow to disable the automatic refresh when doing multiple processing steps• Serialize curve and image styles in HDF5 files• Improve curve readability
0.920 11	• Run computations in a separate process• Add a plugin system: API for third-party extensions• Add a macro-command system• Add an XML-RPC server to allow DataLab remote control

CHAPTER

FOUR

RELEASE NOTES

See DataLab roadmap page for future and past milestones.

4.1 DataLab Version 1.0.1

This major release represents a significant milestone for DataLab with numerous enhancements across all areas. The changes are organized by category for easier navigation.

> **Note**: Version 1.0.1 is a patch release that addresses Windows installer limitations in version 1.0.0, where it was not possible to install DataLab V1.0 alongside earlier major versions (V0.20 and earlier) despite the intended coexistence support. This release contains no new features or functional changes.

4.1.1 User Interface & Workflow

Menu reorganization:

- **New "Create" menu**: Separated object creation functionality from the "File" menu, placed between "File" and "Edit" menus for clearer organization
  - All "File > New [...]" actions moved to "Create" menu
  - **Migration note**: Find signal/image creation actions in the new "Create" menu
- **New "ROI" menu**: Dedicated menu for Region of Interest management, positioned between "Edit" and "Operations" menus
- **New "Annotations" submenu**: Consolidated annotation operations in the "Edit" menu
- **New "Metadata" submenu**: Grouped all metadata operations in the "Edit" menu

Interactive object editing:

- **Interactive object creation**: Creation parameters can be modified after object creation via new "Creation" tab in properties panel
  - Apply changes without creating new objects, preserving subsequent processing
  - Available for parametric generators (Gaussian, sine, etc.)
- **Interactive 1-to-1 processing**: Processing parameters can be adjusted and reapplied via new "Processing" tab
  - Update result objects in-place with modified parameters
  - Only for parametric operations (filters, morphology, etc.)
- **Recompute feature**: New "Recompute" action (Ctrl+R) to quickly reprocess objects with stored parameters

  - Works with single or multiple objects
  - Automatically updates results when source data changes
- **Automatic ROI analysis update**: Analysis results automatically recalculate when ROI is modified
  - Works for all analysis operations (statistics, centroid, etc.)
  - Silent background recomputation for immediate feedback
- **Select source objects**: Navigate to source objects used in processing via new "Edit" menu action
  - Handles all processing patterns (1-to-1, 2-to-1, n-to-1)
  - Shows informative messages if sources no longer exist
- **Processing history**: New "History" tab displays object processing lineage as hierarchical tree
  - Shows complete processing chain from creation to current state
  - Selectable text for documentation purposes

Multi-object property editing:

- Apply property changes to multiple selected objects simultaneously
- Only modified properties are applied, preserving unchanged individual settings
- Typical use case: Adjust LUT boundaries or colormap for multiple images at once

Dialog sizing improvements:

- Processing dialogs now intelligently resize based on main window size
- Never exceed main window dimensions for better user experience

Internal console indicator:

- Status bar indicator shows console status when hidden
- Turns red on errors/warnings to alert users
- Click to open console

4.1.2 New Object Creation Features

Parametric signal generators:

- Linear chirp, logistic function, Planck function
- Generate signals with Poisson noise

Parametric image generators:

- **Checkerboard**: Calibration pattern with configurable square size and light/dark values
- **Sinusoidal grating**: Frequency response testing with independent X/Y spatial frequencies
- **Ring pattern**: Concentric circular rings for radial analysis and PSF testing
- **Siemens star**: Resolution testing with radial spokes and configurable radius
- **2D sinc**: Cardinal sine function for PSF modeling and diffraction simulation
- **2D ramp**: New ramp image generator
- **Poisson noise**: Generate images with Poisson noise distribution

Signal/image creation from operations:

- Create complex-valued signal/image from real and imaginary parts
- Create complex-valued signal/image from magnitude and phase (closes Issue #216)
- Extract phase (argument) information from complex signals or images

4.1.3 Data Processing & Analysis

Signal processing:

- **Enhanced curve fitting**: Significantly improved parameter estimation for all curve types (Gaussian, Lorentzian, Voigt, exponential, sinusoidal, Planckian, asymmetric peaks, CDF)
  - Smarter initial parameter guesses for robust convergence
  - **Locked parameter support**: Lock individual parameters during optimization (requires PlotPy v2.8.0)
- **X-array compatibility checking**: Comprehensive validation for multi-signal operations with automatic interpolation options
  - New configuration setting: Ask user or interpolate automatically
  - Prevents unexpected results from incompatible signal arrays
- **Zero padding enhancements**: Support for prepending and appending zeros, default strategy now "Next power of 2"
- **Ideal frequency domain filter**: "Brick wall filter" for signals (closes Issue #215)
- **Bandwidth at -3dB**: Enhanced to support passband bandwidth
- **Pulse features extraction**: Comprehensive pulse analysis for step and square signals
  - Automated shape recognition and polarity detection
  - Measures amplitude, rise/fall time, FWHM, timing parameters, baseline ranges
- **Frequency domain filters**: Deconvolution and Gaussian filter (closes Issue #189, Issue #205)
- **Coordinate transformations**: Convert to Cartesian/polar coordinates
- **X-Y mode**: Simulate oscilloscope X-Y mode (plot one signal vs. another)
- **Find operations**: First abscissa at y=..., ordinate at x=..., full width at y=...
- **`1/x` operation**: Reciprocal operation with NaN handling for zero denominators

Image processing:

- **2D resampling**: Resample images to new coordinate grids with multiple interpolation methods (closes Issue #208)
- **Convolution**: 2D convolution operation
- **Erase area**: Erase image areas defined by ROI (closes Issue #204)
- **Horizontal/vertical projections**: Sum of pixels along axes (closes Issue #209)
- **Improved centroid computation**: More accurate in challenging cases (truncated/asymmetric images) (see Issue #251)
- **Flip diagonally**: New geometric transformation
- **`1/x` operation**: Reciprocal operation for images

Cross-panel operations:

- **Signals to image conversion**: Combine multiple signals into 2D images

  - Two orientation modes: as rows (spectrograms) or columns (waterfall displays)
  - Optional normalization (Z-score, Min-Max, Maximum)
  - Typical use cases: heatmaps, spectrograms, multi-channel data visualization

Common features:

- **Standard deviation**: Calculate across multiple signals/images (closes Issue #196)
- **Add noise**: Add Gaussian, Poisson, or uniform noise (closes Issue #201)
- **Add metadata**: Add custom metadata with pattern support (`{title}`, `{index}`, etc.) and type conversion

4.1.4 ROI & Annotation Management

ROI features:

- **ROI clipboard operations**: Copy/paste ROIs between objects
- **ROI import/export**: Save/load ROIs as JSON files
- **Individual ROI removal**: Remove ROIs selectively via "Remove" submenu
- **ROI title editing**: Set titles during interactive creation and in confirmation dialog
- **Create ROI grid**: Generate grid of ROIs with configurable rows, columns, and spacing
  - Import/export grid configurations
  - Preview before creation
- **Inverse ROI logic**: Select area outside defined shapes (images only)
- **Coordinate-based ROI creation**: Manual input of coordinates for rectangular and circular ROIs
- **Multi-object ROI editing**: Edit ROIs on multiple objects simultaneously

Annotation features:

- **Annotation clipboard**: Copy/paste annotations between objects
- **Edit annotations**: Interactive editor dialog with PlotPy tools
- **Import/export annotations**: Save/load as .dlabann JSON files with versioning
- **Delete annotations**: Remove from single or multiple objects with confirmation
- **Annotations independent from ROI**: Can coexist on same object

Detection with ROI creation:

- All 7 detection algorithms now support automatic ROI creation:
  - Peak detection, contour shape, blob detection (DOG/DOH/LOG/OpenCV), Hough circle
- **ROI geometry choice**: Rectangular or circular ROIs
- **2D peak detection**: Option to choose ROI geometry (closes related requirements)

4.1.5 Visualization & Display

Performance & display limits:

- **Configurable result display limits**: Prevent UI freezing with large result sets
  - `max_shapes_to_draw` (default: 1,000), `max_cells_in_label` (default: 100), `max_cols_in_label` (default: 15)
  - Settings documented with performance implications
- **Faster contour rendering**: Over 5x performance improvement for contour display
- **Signal rendering optimization**: Smart linewidth clamping for large datasets
  - New setting: "Line width performance threshold" (default: 1,000 points)
  - Prevents 10x slowdown from Qt raster engine limitation

Result visualization:

- **Merged result labels**: All analysis results consolidated in single read-only label
  - Reduces visual clutter, auto-updates, horizontally divided results
- **Result label visibility control**: Toggle visibility via Properties panel checkbox
  - Default visibility configurable in Settings
- **Results group organization**: Plot results automatically organized in dedicated "Results" group
- **Comprehensive result titles**: Include source object identifiers (e.g., "FWHM (s001, s002, s003)")
- **Individual result deletion**: Remove results selectively via Analysis menu
- **Customizable shape/marker styles**: Four new style configuration buttons in Settings
  - Separate styles for signals and images
  - Interactive editor with comprehensive options
  - Persistent configuration with refresh on change

Enhanced profile extraction:

- Click directly on X/Y profile plots to switch extraction direction
- No need to open parameters dialog for direction changes
- Improves workflow efficiency (closes Issue #156)

DateTime signal support:

- Automatic datetime detection in CSV files
- **Datetime axis formatting**: Human-readable timestamps on X-axis
- **Configurable formats**: Separate formats for standard and sub-second units
- Supports various time units (seconds, milliseconds, microseconds, minutes, hours)
- Closes Issue #258

Settings:

- **Autoscale margins**: Configurable margins (0-50%) for signal/image plots
- **Lock image aspect ratio**: Option for 1:1 aspect ratio (default: use physical pixel size) (closes Issue #244)
- **Show console on error**: Configurable behavior (default: off)

Image extent parameters:

- New "Extent" group box showing computed Xmin, Xmax, Ymin, Ymax
- Automatically calculated from origin, pixel spacing, and dimensions

4.1.6 Import/Export & File Handling

New file format support:

- **FT-Lab signals and images**: CEA binary formats (.sig, .ima) (closes Issue #211)
- **Coordinated text files**: Real and complex-valued images with error images (similar to Matris format)
  - Automatic NaN handling, metadata with units and labels

Enhanced HDF5 support:

- **Custom file extensions**: Intelligent HDF5 detection by content (not just extension)
  - Extension-based detection for dialogs, content-based for drag-and-drop
  - "All files (*)" option in file dialogs
- **HDF5 Browser improvements**: Default collapsed tree view for better navigation
- **Workspace clearing options**: Configurable behavior with "Ignore" option (closes Issue #146)

Text file improvements:

- **CSV delimiter handling**: Better support for various whitespace separators
- **Locale decimal separator**: Support for comma as decimal separator (closes Issue #124)
- **Encoding error tolerance**: Ignore errors for files with special characters
- **Header detection**: Automatic detection and skipping of data headers

Other I/O features:

- **Save to directory**: New feature (closes Issue #227)
- **Open from directory**: Recursively open multiple files with folder drag-and-drop support
- **File ordering**: Consistent alphabetical sorting across platforms

4.1.7 Advanced Features

Non-uniform coordinate support:

- Images now support non-uniform pixel spacing
- "Set uniform coordinates" feature for conversion
- **Polynomial calibration**: Up to cubic order for X, Y, Z axes
  - Creates non-uniform coordinates for X/Y, transforms values for Z
- HDF5 and text file formats preserve non-uniform information

Group management:

- **Panel-specific short IDs**: `gs` prefix for signals, `gi` prefix for images
  - Avoids ambiguity in cross-panel operations
- Fixed group numbering for new groups

Configuration:

- **Version-specific folders**: Major version coexistence (`.DataLab_v1`, `.DataLab_v2`, etc.)
  - Allows v0.x and v1.x to run simultaneously

Public API & Remote Control:

- Enhanced metadata handling with function name context
- `add_group`, `add_signal`, `add_image` methods with `group_id` and `set_current` arguments
- `get_object_uuids` with optional `group` filter
- Multiple API improvements for better programmability

Processing infrastructure (for developers):

- New `@computation_function` decorator for computation functions
- Renamed computation functions (removed "compute_" prefix)
- Refactored `BaseProcessor` methods with clear naming:
  - `compute_1_to_1`, `compute_1_to_0`, `compute_1_to_n`, `compute_n_to_1`, `compute_2_to_1`
- **No backward compatibility maintained** for these internal changes (closes Issue #180)

4.1.8 Bug Fixes

Performance fixes:

- **Switching between images with many results**: Dramatic improvement (66s → <1ms) when navigating images with hundreds of shapes
- **Plot cleanup robustness**: Fixed errors when removing analysis results
- **Critical fix**: Result labels now properly excluded from cleanup to prevent disappearance

Cross-panel & group handling:

- **Cross-panel computation groups**: Fixed inconsistent group organization for image-to-signal operations
- **File import ordering**: Consistent alphabetical sorting across all platforms

Action state updates:

- Fixed action enable states not updating after annotation/metadata operations
- UI now immediately reflects current object state

Result management:

- Fixed result label deletion to permanently remove associated metadata
- Fixed duplicate results parameter metadata cleanup
- **Result coordinate fixes**: Corrected shifted results on images with ROIs and shifted origin (Issue #106)
- **Profile extraction indices**: Fixed wrong indices with ROI (Issue #107)

ROI-related fixes:

- Fixed multi-image ROI extraction not saving ROI in first object (Issue #120)
- Fixed AttributeError when extracting multiple ROIs on single image with multiple selections (Issue #121)
- Fixed mask refresh issues (Issue #122, Issue #123)

- Fixed ROI editor on multiple signals/images (Issue #135)
- Fixed ROI clearing affecting only first image (Issue #160)
- Fixed ROI editor showing first instead of last image (Issue #158)

Text Import Wizard:

- Fixed preservation of user-defined titles and units (Issue #239)
- Fixed preservation of data types (Issue #240)
- Fixed comma decimal separator support (Issue #186)
- Fixed trailing delimiter issue (Issue #238)

Curve fitting & analysis:

- Improved initial frequency estimate for sinusoidal fitting
- Fixed parameter display formatting for extreme values
- Fixed FWHM computation exception handling
- Fixed curve marker style changing unexpectedly (Issue #184)
- Fixed hard crash on zero signal with curve stats tool (Issue #233)

Image handling:

- Fixed aspect ratio not updating when switching images
- Fixed shape unpacking issues (Issue #246, Issue #247)
- Fixed colormaps not stored in metadata (PlotPy v2.6.3+ issue) (Issue #138)
- Fixed amplitude calculation for non-integer data types

Signal processing:

- Fixed ifft1d x-axis computation when shift=False (Issue #241)
- Fixed moving median crash on Linux with mirror mode (Issue #117) - SciPy bug
- Fixed magnitude spectrum with logarithmic scale (Issue #169)
- Fixed pairwise operation mode for asymmetric functions (Issue #157)

Analysis & results:

- Fixed NaN value handling in statistics and normalization (Issue #141, Issue #152, Issue #153)
- Fixed analysis results kept from original after processing (Issue #136)
- Fixed duplicate results with no ROI defined
- Fixed "One curve per result title" mode ignoring ROIs (Issue #132)
- Added result validation for array-like results

Plot & visualization:

- Fixed profile plots not refreshing when moving/resizing (Issue #172) - PlotPy fix
- Fixed empty average profile display outside image area (Issue #168) - PlotPy fix
- Fixed average profile extraction ValueError with oversized rectangle (Issue #144)
- Disabled generic "Axes" tab in parameter dialogs

Other fixes:

- Fixed proxy `add_object` method not supporting metadata (Issue #111)
- Fixed RemoteClient method calls without optional arguments (Issue #113)
- Fixed KeyError when removing group after opening HDF5 (Issue #116)
- Fixed KeyError in "View in new window" with multiple images after HDF5 open (Issue #159)
- Fixed long object titles display (Issue #128)
- Fixed file name titles showing relative paths (Issue #165)
- Fixed unexpected group names in "Open from directory" (Issue #177)
- Fixed one group per folder expectation (Issue #163)
- Fixed unsupported files in recursive loading (Issue #164)

Removed features:

- Removed "Use reference image LUT range" setting (confusing behavior)
  - **Migration**: Use multi-selection property editing instead

4.1.9 Security Fixes

- **CVE-2023-4863**: Fixed vulnerability in opencv-python-headless
  - Updated minimum requirement from 4.5.4.60 to 4.8.1.78
  - See DataLab security advisory

4.1.10 Other Changes

- Bumped minimum `plotpy` requirement to V2.8
- Bumped minimum `guidata` requirement to V3.13
- Using new `guidata` translation utility based on `babel`
- Python 3.13 now supported (via scikit-image V0.25)

4.2 DataLab Version 0.20.0

New features and enhancements:

- ANDOR SIF Images:
  - Added support for background images in ANDOR SIF files
  - This closes Issue #178 - Add support for ANDOR SIF files with background image
- Array editor (results, signal and image data, ...):
  - New "Copy all" button in the array editor dialog box, to copy all the data in the clipboard, including row and column headers
  - New "Export" button in the array editor dialog box, to export the data in a CSV file, including row and column headers
  - New "Paste" button in the array editor dialog box, to paste the data from the clipboard into the array editor (this feature is not available for read-only data, such as analysis results)

  - The features above require guidata v3.9.0 or later
  - This closes Issue #174, Issue #175 and Issue #176
- Fourier analysis features ("Processing" menu):
  - New "Zero padding" feature
  - Implementation for signals:
    * Choose a zero padding strategy (Next power of 2, Double the length, Triple the length, Custom length)
    * Or manually set the zero padding length (if "Custom length" is selected)
  - Implementation for images:
    * Choose a zero padding strategy (Next power of 2, Next multiple of 64, Custom length)
    * Or manually set the zero padding row and column lengths (if "Custom length" is selected)
    * Set the position of the zero padding (bottom-right, centered)
  - This closes Issue #170 - Fourier analysis: add zero padding feature for signals and images
- Region of Interest (ROI) editor:
  - This concerns the "Edit Regions of Interest" feature for both signals and images
  - New behavior:
    * Signals: the range ROI selection tool is now active by default, and the user can select right away the range of the signal to be used as a ROI
    * Images: the rectangular ROI selection tool is now active by default, and the user can select right away the rectangular ROI to be used as a ROI
    * This closes Issue #154 - ROI editor: activate ROI selection tool by default, so that the user can select right away the area to be used as a ROI
  - Added the "Select tool" to editor's toolbar, to allow the user to switch between the "Select" and "Draw" tools easily without having to use the plot toolbar on the top of the window
- Signal processing features ("Processing" menu):
  - New "X-Y mode" feature: this feature simulates the behavior of the X-Y mode of an oscilloscope, i.e. it allows to plot one signal as a function of another signal (e.g. X as a function of Y)
  - New abscissa and ordinate find features:
    * "First abscissa at y=…" feature: this feature allows to find the first abscissa value of a signal at a given y value (e.g. the abscissa value of a signal at y=0)
    * "Ordinate at x=…" feature: this feature allows to find the ordinate value of a signal at a given x value (e.g. the ordinate value of a signal at x=0)
    * Each feature has its own dialog box, which allows to set the y or x value to be used for the search with a slider or a text box
    * This closes Issue #125 and Issue #126
  - New full width at given y feature:
    * The "Full width at y=…" feature allows to find the full width of a signal at a given y value (e.g. the full width of a signal at y=0)
    * A specific dialog box allows to set the y value to be used for the search with a slider or a text box
    * This closes Issue #127

- Public API (local or remote):
  - Add `group_id` and `set_current` arguments to `add_signal`, `add_image` and `add_object` methods:
    * This concerns the `LocalProxy`, `AbstractDLControl`, `RemoteClient`, `RemoteServer` and `DLMainWindow` classes
    * `group_id` argument allows to specify the group ID where the signal or image should be added (if not specified, the signal or image is added to the current group)
    * `set_current` argument allows to specify if the signal or image should be set as current after being added (default is `True`)
    * This closes Issue #151 - Public API: add a keyword `group_id` to `add_signal` and `add_image`

4.3 DataLab Version 0.19.2

Bug fixes:

- Fixed Issue #172 - Image profiles: when moving/resizing image, profile plots are not refreshed (fixed in PlotPy v2.7.4)
- Fixed Issue #173 - Phase spectrum: add unit (degree) and function reference (`numpy.angle`) to the documentation
- Fixed Issue #177 - "Open from directory" feature: unexpected group name (a group named "." is created instead of the root folder name)
- Fixed Issue #169 - Signal / Fourier analysis: magnitude spectrum feature does not work as expected with logarithmic scale enabled
- Fixed Issue #168 - Average profile visualization: empty profile is displayed when the target rectangular area is outside the image area (this has been fixed upstream, in PlotPy v2.7.4, and so requires the latest version of PlotPy)

4.4 DataLab Version 0.19.1

Bug fixes:

- Pairwise operation mode:
  - Fixed an unexpected behavior when using the pairwise operation mode with functions that take a single second operand (e.g. for images: difference, division, arithmetic operations, and flatfield correction)
  - If only one set of operands was selected in a single group, a warning message was displayed "In pairwise mode, you need to select objects in at least two groups.", which is correct for functions that are symmetric (e.g. addition, multiplication, etc.), but not for functions that are not symmetric (e.g. difference, division, etc.).
  - This is now fixed: the warning message is only displayed for functions that are symmetric (e.g. addition, multiplication, etc.).
  - This closes Issue #157 - Pairwise operation mode: unexpected behavior with functions that take a single second operand
- Fixed Issue #152 - Ignore `nan` values for image normalization, flatfield correction, offset correction, and centroid computation
- Fixed Issue #153 - Ignore `nan` values for signal normalization and statistics computations (both analysis result and interactive tool)

- Fixed Issue #158 - When editing ROI of a list of images, the first image of the selection is shown (instead of the last as in the image panel)
- Fixed Issue #159 - When selecting multiple images just after opening an HDF5 file, the "View in a new window" feature does not work (`KeyError` exception)
- Fixed Issue #160 - When selecting multiple images and clearing ROI in ROI editor, only the first image is affected
- Fixed Issue #161 - Refresh image items only if necessary (when editing ROI, pasting/deleting metadata)
- Fixed Issue #162 - View in a new window: when displaying multiple images, the item list panel should be visible
- Fixed Issue #163 - Open from directory: expected one group per folder when loading multiple files
- Fixed Issue #164 - Open from directory: unsupported files should be ignored when loading files recursively, to avoid warning popup dialog boxes
- Fixed Issue #165 - When opening a file, the default signal/image title must be set to the file name, instead of the relative path to the file name

4.5 DataLab Version 0.19.0

New features and enhancements:

- Image operation features ("Operations" menu):
  - Renamed "Rotation" submenu to "Flip or rotation"
  - New "Flip diagonally" feature
- Signal processing features ("Processing" menu):
  - New "Convert to Cartesian coordinates" feature
  - New "Convert to polar coordinates" feature
- Signal analysis features ("Analysis" menu):
  - Renamed "X values at min/max" to "Abscissa of the minimum and maximum"
  - New "Abscissa at y=…" feature
- New "Open from directory" feature:
  - This feature allows to open multiple files from a directory at once, recursively (only the files with the supported extensions by the current panel are opened)
  - Add "Open from directory" action to the "File" menu for both Signal and Image panels
  - Add support for folders when dropping files in the Signal and Image panels
- Add `1/x` operation to the "Operations" menu for both Signal and Image panels:
  - This feature relies on the `numpy.reciprocal` function, and handles the case where the denominator is zero by catching warnings and replacing the `np.inf` values with `np.nan` values
  - Add `compute_inverse` method for image and signal processors
  - This closes Issue #143 - New feature: `1/x` for signals and images
- Public API (local or remote):
  - Add `add_group` method with `title` and `select` arguments to create a new group in a data panel (e.g. Signal or Image panel) and eventually select it after creation:

    * Method was added to the following classes: `AbstractDLControl`, `BaseDataPanel` and `RemoteClient`
    * This closes the following issues:
      · Issue #131 - `BaseDataPanel.add_group`: add `select` argument
      · Issue #47 - Remote proxy / Public API: add `add_group` method
  - `AbstractDLControl.get_object_uuids`: add an optional `group` argument (group ID, title or number) to eventually filter the objects by group (this closes Issue #130)
- When opening an HDF5 file, the confirmation dialog box asking if current workspace should be cleared has a new possible answer "Ignore":
  - Choosing "Ignore" will prevent the confirmation dialog box from being displayed again, and will choose the current setting (i.e. clear or not the workspace) for all subsequent file openings
  - Added a new "Clear workspace before loading HDF5 file" option in the "Settings" dialog box, to allow the user to change the current setting (i.e. clear or not the workspace) for all subsequent file openings
  - Added a new "Ask before clearing workspace" option in the "Settings" dialog box, to allow the user to disable or re-enable the confirmation dialog box asking if current workspace should be cleared when opening an HDF5 file
  - This closes Issue #146 - Ask before clearing workspace when opening HDF5 file: add "Ignore" option to prevent dialog from being displayed again
- Object and group title renaming:
  - Removed "Rename group" feature from the "Edit" menu and context menu
  - Added "Rename object" feature to the "Edit" menu and context menu, with F2 shortcut, to rename the title of the selected object or group
  - This closes Issue #148 - Rename signal/image/group title by pressing F2
- Region of Interest editor:
  - Regrouped the graphical actions (new rectangular ROI, new circular ROI, new polygonal ROI) in a single menu "Graphical ROI"
  - Added new "Coordinate-based ROI" menu to create a ROI using manual input of the coordinates:
    * For signals, the ROI is defined by the start and end coordinates
    * For images:
      · The rectangular ROI is defined by the top-left and bottom-right coordinates
      · The circular ROI is defined by the center and radius coordinates
      · The polygonal ROI is not supported yet
    * This closes Issue #145 - ROI editor: add manual input of the coordinates

Bug fixes:

- Fixed Issue #141 - Image analysis: mask `nan` values when computing statistics, for example
- Fixed Issue #144 - Average profile extraction: `ValueError` when selection rectangle is larger than the image

4.6 DataLab Version 0.18.2

General information:

- Python 3.13 is now supported, since the availability of the scikit-image V0.25 (see Issue #104 - Python 3.13: `KeyError: 'area_bbox'`)

Enhancements:

- Added new "Keep results after computation" option in "Processing" section:
  - Before this change, when applying a processing feature (e.g. a filter, a threshold, etc.) on a signal or an image, the analysis results were removed from the object
  - This new option allows to keep the analysis results after applying a processing feature on a signal or an image. Even if the analysis results are not updated, they might be relevant in some use cases (e.g. when using the 2D peak detection feature on an image, and then applying a filter on the image, or summing two images, etc.)

Bug fixes:

- Fixed Issue #138 - Image colormaps were no longer stored in metadata (and serialized in HDF5 files) since PlotPy v2.6.3 (this commit, specifically: PlotPyStack/PlotPy@a37af8a)
- Fixed Issue #137 - Arithmetic operations and signal interpolation: dialog box with parameters is not displayed
- Fixed Issue #136 - When processing a signal or an image, the analysis result is kept from original object
  - Before this fix, when processing a signal or an image (e.g. when applying a filter, a threshold, etc.), the analysis result was kept from the original object, and was not updated with the new data. Thus the analysis result was not meaningful anymore, and was misleading the user.
  - This is now fixed: the analysis result is now removed when processing a signal or an image. However it is not recalculated automatically, because there is no way to know which analysis result should be recalculated (e.g. if the user has applied a filter, should the FWHM be recalculated?) - besides, the current implementation of the analysis features does not allow to recalculate the analysis results automatically when the data is modified. The user has to recalculate the analysis results manually if needed.
- Fixed Issue #132 - Plot analysis results: "One curve per result title" mode ignores ROIs
  - Before this fix, the "One curve per result title" mode was ignoring ROIs, and was plotting the selected result for all objects (signals or images) without taking into account the ROI defined on the objects
  - This is now fixed: the "One curve per result title" mode now takes into account the ROI defined on the objects, and plots the selected result for each object (signal or image) and for each ROI defined on the object
- Fixed Issue #128 - Support long object titles in Signal and Image panels
- Fixed Issue #133 - Remove specific analysis results from metadata clipboard during copy operation
- Fixed Issue #135 - Allow to edit ROI on multiple signals or images at once
  - Before this fix, the ROI editor was disabled when multiple signals or images were selected
  - This is now fixed: the ROI editor is now enabled when multiple signals or images are selected, and the ROI is applied to all selected signals or images (only the ROI of the first selected signal or image is taken into account)
  - This new behavior is consistent with the ROI extraction feature, which allows to extract the ROI on multiple signals or images at once, based on the ROI defined on the first selected signal or image
- Image ROI features:

  - Fixed Issue #120 - ROI extraction on multiple images: defined ROI should not be saved in the first selected object. The design choice is to save the defined ROI neither in the first nor in any of the selected objects: the ROI is only used for the extraction, and is not saved in any object
  - Fixed Issue #121 - `AttributeError` when extracting multiple ROIs on a single image, if more than one image is selected
  - Fixed Issue #122 - Image masks are not refreshed when removing metadata except for the active image
  - Fixed Issue #123 - Image masks are not refreshed when pasting metadata on multiple images, except for the last image
- Text and CSV files:
  - Enhance text file reading by detecting data headers (using a list of typical headers from scientific instruments) and by allowing to skip the header when reading the file
  - Ignore encoding errors when reading files in both open feature and import wizard, hence allowing to read files with special characters without raising an exception
  - Fixed Issue #124 - Text files: support locale decimal separator (different than `.`)
- Signal analysis features: fixed duplicate results when no ROI is defined
- Fixed Issue #113 - Call to `RemoteClient.open_h5_files` (and `import_h5_file`) fails without passing the optional arguments
- Fixed Issue #116 - `KeyError` exception when trying to remove a group after opening an HDF5 file

4.7 DataLab Version 0.18.1

Enhancements:

- FWHM computation now raises an exception when less than two points are found with zero-crossing method
- Improved result validation for array-like results by checking the data type of the result

Bug fixes:

- Fixed Issue #106 - Analysis: coordinate shifted results on images with ROIs and shifted origin
- Fixed Issue #107 - Wrong indices when extracting a profile from an image with a ROI
- Fixed Issue #111 - Proxy `add_object` method does not support signal/image metadata (e.g. ROI)
- Test data plugin / "Create 2D noisy gauss image": fixed amplitude calculation in `datalab.tests.data.create_2d_random` for non-integer data types

Documentation:

- Fixed path separators in plugin directory documentation
- Corrected left and right area descriptions in workspace documentation
- Updated Google style link in contributing guidelines
- Fixed various French translations in the documentation

4.8 DataLab Version 0.18.0

General information:

- PlotPy v2.7 is required for this release.
- Dropped support for Python 3.8.
- Python 3.13 is not supported yet, due to the fact that some dependencies are not compatible with this version.

New features and enhancements:

- New operation mode feature:
  - Added "Operation mode" feature to the "Processing" tab in the "Settings" dialog box
  - This feature allows to choose between "single" and "pairwise" operation modes for all basic operations (addition, subtraction, multiplication, division, etc.):
    - "Single" mode: single operand mode (default mode: the operation is done on each object independently)
    - "Pairwise" mode: pairwise operand mode (the operation is done on each pair of objects)
  - This applies to both signals and images, and to computations taking *N* inputs
  - Computations taking *N* inputs are the ones where:
    - *N(>=2)* objects in give *N* objects out
    - *N(>=1)* object(s) + 1 object in give N objects out
- New ROI (Region Of Interest) features:
  - New polygonal ROI feature
  - Complete redesign of the ROI editor user interfaces, improving ergonomics and consistency with the rest of the application
  - Major internal refactoring of the ROI system to make it more robust (more tests) and easier to maintain
- Implemented Issue #102 - Launch DataLab using `datalab` instead of `datalab`. Note that the `datalab` command is still available for backward compatibility.
- Implemented Issue #101 - Configuration: set default image interpolation to anti-aliasing (`5` instead of `0` for nearest). This change is motivated by the fact that a performance improvement was made in PlotPy v2.7 on Windows, which allows to use anti-aliasing interpolation by default without a significant performance impact.
- Implemented Issue #100 - Use the same installer and executable on Windows 7 SP1, 8, 10, 11. Before this change, a specific installer was required for Windows 7 SP1, due to the fact that Python 3.9 and later versions are not supported on this platform. A workaround was implemented to make DataLab work on Windows 7 SP1 with Python 3.9.

Bug fixes:

- Fixed Issue #103 - `proxy.add_annotations_from_items`: circle shape color seems to be ignored.

4.9 DataLab Version 0.17.1

PlotPy v2.6.2 is required for this release.

New features and enhancements:

- Image View:
  - Before this release, when selecting a high number of images (e.g. when selecting a group of images), the application was very slow because all the images were displayed in the image view, even if they were all superimposed on the same image
  - The workaround was to enable the "Show first only" option
  - Now, to improve performance, if multiple images are selected, only the last image of the selection is displayed in the image view if this last image has no transparency and if the other images are completely covered by this last image
- Clarification: action "Show first only" was renamed to "Show first object only", and a new icon was added to the action
- API: added `width` and `height` properties to `ImageObj` class (returns the width and height of the image in physical units)
- Windows launcher "start.pyw": writing a log file "datalab_error.log" when an exception occurs at startup

Bug fixes:

- Changing the color theme now correctly updates all DataLab's user interface components without the need to restart the application

Other changes:

- OpenCV is now an optional dependency:
  - This change is motivated by the fact that the OpenCV conda package is not maintained on Windows (at least), which leads to an error when installing DataLab with conda
  - When OpenCV is not installed, only the "OpenCV blob detection" feature won't work, and a warning message will be displayed when trying to use this feature

4.10 DataLab Version 0.17.0

PlotPy v2.6 is required for this release.

New features and enhancements:

- Menu "Computing" was renamed to "Analysis" for both Signal and Image panels, to better reflect the nature of the features in this menu
- Regions Of Interest (ROIs) are now taken into account everywhere in the application where it makes sense, and not only for the old "Computing" menu (now "Analysis") features. This closes Issue #93. If a signal or an image has an ROI defined:
  - Operations are done on the ROI only (except if the operation changes the data shape, or the pixel size for images)
  - Processing features are done on the ROI only (if the destination object data type is compatible with the source object data type, which excludes thresholding, for instance)
  - Analysis features are done on the ROI only, like before

- As a consequence of previous point, and for clarity:
  - The "Edit Regions of interest" and "Remove all Regions of interest" features have been moved from the old "Computing" (now "Analysis") menu to the "Edit" menu where all metadata-related features are located
  - The "Edit Regions of interest" action has been added to both Signal and Image View vertical toolbars (in second position, after the "View in a new window" action)
- Following the bug fix on image data type conversion issues with basic operations, a new "Arithmetic operation" feature has been added to the "Operations" menu for both Signal and Image panels. This feature allows to perform linear operations on signals and images, with the following operations:
  - Addition: `obj3 = (obj1 + obj2) * a + b`
  - Subtraction: `obj3 = (obj1 - obj2) * a + b`
  - Multiplication: `obj3 = (obj1 * obj2) * a + b`
  - Division: `obj3 = (obj1 / obj2) * a + b`
- Improved "View in a new window" and "ROI editor" dialog boxes size management: default size won't be larger than DataLab's main window size
- ROI editor:
  - Added toolbars for both Signal and Image ROI editors, to allow to zoom in and out, and to reset the zoom level easily
  - Rearranged the buttons in the ROI editor dialog box for better ergonomics and consistency with the Annotations editor ("View in a new window" dialog box)
- Application color theme:
  - Added support for color theme (auto, light, dark) in the "Settings" dialog box
  - The color theme is applied without restarting the application

Bug fixes:

- Intensity profile / Segment profile extraction:
  - When extracting a profile on an image with a ROI defined, the associated PlotPy feature show a warning message ('UserWarning: Warning: converting a masked element to nan.') but the profile is correctly extracted and displayed, with NaN values where the ROI is not defined.
  - NaN values are now removed from the profile before plotting it
- Simple processing features with a one-to-on mapping with a Python function (e.g. `numpy.absolute`, `numpy.log10`, etc.) and without parameters: fix result object title which was systematically ending with "|" (the character that usually precedes the list of parameters)
- Butterworth filter: fix cutoff frequency ratio default value and valid range
- Fix actions refresh issue in Image View vertical toolbar:
  - When starting DataLab with the Signal Panel active, switching to the Image View was showing "View in a new window" or "Edit Regions of interest" actions enabled in the vertical toolbar, even if no image was displayed in the Image View
  - The Image View vertical toolbar is now correctly updated at startup
- View in a new window: cross section tools (intensity profiles) stayed disabled unless the user selected an image through the item list - this is now fixed
- Image View: "Show contrast panel" toolbar button was not enabled at startup, and was only enabled when at least one image was displayed in the Image View - it is now always enabled, as expected

- Image data type conversion:
  - Previously, the data type conversion feature was common to signal and image processing features, i.e. a simple conversion of the data type using NumPy's `astype` method
  - This was not sufficient for image processing features, in particular for integer images, because even if the result was correct from a numerical point of view, underflow or overflow could be legitimately seen as a bug from a mathematical point of view
  - The image data type conversion feature now relies on the internal `clip_astype` function, which clips the data to the valid range of the target data type before converting it (in the case of integer images)
- Image ROI extraction issues:
  - Multiple regressions were introduced in version 0.16.0:
    * Single circular ROI extraction was not working as expected (a rectangular ROI was extracted, with unexpected coordinates)
    * Multiple circular ROI extraction lead to a rectangular ROI extraction
    * Multiple ROI extraction was no longer cropping the image to the overall bounding box of the ROIs
  - These issues are now fixed, and unit tests have been added to prevent regressions:
    * An independent test algorithm has been implemented to check the correctness of the ROI extraction in all cases mentioned above
    * Tests cover both single and multiple ROI extraction, with circular and rectangular ROIs
- Overflow and underflow issues in some operations on integer images:
  - When processing integer images, some features were causing overflow or underflow issues, leading to unexpected results (correct results from a numerical point of view, but not from a mathematical point of view)
  - This issue only concerned basic operations (addition, subtraction, multiplication, division, and constant operations) - all the other features were already working as expected
  - This is now fixed as result output are now floating point images
  - Unit tests have been added to prevent regressions for all these operations

4.11 DataLab Version 0.16.4

This is a minor maintenance release.

Bug fixes:

- Requires PlotPy v2.4.1 or later to fix the following issues related to the contrast adjustment feature:
  - A regression was introduced in an earlier version of PlotPy: levels histogram was no longer removed from contrast adjustment panel when the associated image was removed from the plot
  - This is now fixed: when an image is removed, the histogram is removed as well and the contrast panel is refreshed (which was not the case even before the regression)
- Ignore `AssertionError` in *config_unit_test.py* when executing test suite on WSL

Documentation:

- Fix class reference in `Wrap11Func` documentation

4.12 DataLab Version 0.16.3

Bug fixes:

- Fixed Issue #84 - Build issues with V0.16.1: `signal` name conflict, ...
  - This issue was intended to be fixed in version 0.16.2, but the fix was not complete
  - Thanks to @rolandmas for reporting the issue and for the help in investigating the problem and testing the fix
- Fixed Issue #85 - Test data paths may be added multiple times to `datalab.utils.tests.TST_PATH`
  - This issue is related to Issue #84
  - Adding the test data paths multiple times to `datalab.utils.tests.TST_PATH` was causing the test data to be loaded multiple times, which lead to some tests failing (a simple workaround was added to V0.16.2: this issue is now fixed)
  - Thanks again to @rolandmas for reporting the issue in the context of the Debian packaging
- Fixed Issue #86 - Average of N integer images overflows data type
- Fixed Issue #87 - Image average profile extraction: `AttributeError` when trying to edit profile parameters
- Fixed Issue #88 - Image segment profile: point coordinates inversion

4.13 DataLab Version 0.16.2

This release requires PlotPy v2.4.0 or later, which brings the following bug fixes and new features:

- New constrast adjustment features and bug fixes:
  - New layout: the vertical toolbar (which was constrained in a small area on the right side of the panel) is now a horizontal toolbar at the top of the panel, beside the title
  - New "Set range" button: allows the user to set manually the minimum and maximum values of the histogram range
  - Fixed histogram update issues when no image was currently selected (even if the an image was displayed and was selected before)
  - Histogram range was not updated when either the minimum or maximum value was set using the "Minimum value" or "Maximum value" buttons (which have been renamed to "Min." and "Max." in this release)
  - Histogram range was not updated when the "Set full range" button was clicked, or when the LUT range was modified using the "Scales / LUT range" form in "Properties" group box
- Image view context menu: new "Reverse X axis" feature

Minor new features and enhancements:

- Image file types:
  - Added native support for reading .SPE, .GEL, .NDPI and .REC image files
  - Added support for any `imageio`-supported file format through configuration file (entry `imageio_formats` may be customized to complement the default list of supported formats: see documentation for more details)

Bug fixes:

- Image Fourier analysis:
  - Fixed logarithmic scale for the magnitude spectrum (computing dB instead of natural logarithm)

  - Fixed PSD computation with logarithmic scale (computing dB instead of natural logarithm)
  - Updated the documentation to explicitly mention that the logarithmic scale is in dB
- Fixed Issue #82 - Macros are not renamed in DataLab after exporting them to Python scripts
- `ResultProperties` object can now be added to `SignalObj` or `ImageObj` metadata even outside a Qt event loop (because the label item is no longer created right away)
- Progress bar is now automatically closed as expected when an error occurrs during a long operation (e.g. when opening a file)
- Difference, division…: dialog box for the second operand selection was allowing to select a group (only a signal or an image should be selected)
- When doing an operation which involves an object (signal or image) with higher order number than the current object (e.g. when subtracting an image with an image from a group below the current image), the resulting object's title now correctly refers to the order numbers of the objects involved in the operation (e.g., to continue with the subtraction example mentioned above, the resulting object's title was previously referring to the order number before the insertion of the resulting image)
- Added support for additional test data folder thanks to the `DATALAB_DATA` environment variable (useful for testing purposes, and especially in the context of Debian packaging)

4.14 DataLab Version 0.16.1

Since version 0.16.0, many validation functions have been added to the test suite. The percentage of validated compute functions has increased from 37% to 84% in this release.

NumPy 2.0 support has been added with this release.

Minor new features and enhancements:

- Signal and image moving average and median filters:
  - Added "Mode" parameter to choose the mode of the filter (e.g. "reflect", "constant", "nearest", "mirror", "wrap")
  - The default mode is "reflect" for moving average and "nearest" for moving median
  - This allows to handle edge effects when filtering signals and images

Bug fixes:

- Fixed Canny edge detection to return binary image as `uint8` instead of `bool` (for consistency with other image processing features)
- Fixed Image normalization: lower bound was wrongly set for `maximum` method
- Fixed `ValueError` when computing PSD with logarithmic scale
- Fixed Signal derivative algorithm: now using `numpy.gradient` instead of a custom implementation
- Fixed SciPy's `cumtrapz` deprecation: use `cumulative_trapezoid` instead
- Curve selection now shows the individual points of the curve (before, only the curve line width was broadened)
- Windows installer: add support for unstable releases (e.g., 0.16.1.dev0), thus allowing to easily install the latest development version of DataLab on Windows
- Fixed Issue #81 - When opening files, show progress dialog only if necessary
- Fixed Issue #80 - Plotting results: support for two use cases

  - The features of the "Analysis" menu produce *results* (scalars): blob detection (circle coordinates), 2D peak detection (point coordinates), etc. Depending on the feature, result tables are displayed in the "Results" dialog box, and the results are also stored in the signal or image metadata: each line of the result table is an individual result, and each column is a property of the result - some results may consist only of a single individual result (e.g., image centroid or curve FHWM), while others may consist of multiple individual results (e.g., blob detection, contour detection, etc.).
  - Before this change, the "Plot results" feature only supported plotting the first individual result of a result table, as a function of the index (of the signal or image objects) or any of the columns of the result table. This was not sufficient for some use cases, where the user wanted to plot multiple individual results of a result table.
  - Now, the "Plot results" feature supports two use cases:
    * "One curve per result title": Plotting the first individual result of a result table, as before
    * "One curve per object (or ROI) and per result title": Plotting all individual results of a result table, as a function of the index (of the signal or image objects) or any of the columns of the result table
  - The selection of the use case is done in the "Plot results" dialog box
  - The default use case is "One curve per result title" if the result table has only one line, and "One curve per object (or ROI) and per result title" otherwise

4.15 DataLab Version 0.16.0

New features and enhancements:

- Major user interface overhaul:
  - The menu bar and toolbars have been reorganized to make the application more intuitive and easier to use
  - Operations and processing features have been regrouped in submenus
  - All visualization-related actions are now grouped in the plot view vertical toolbar
  - Clarified the "Annotations" management (new buttons, toolbar action…)
- New validation process for signal and image features:
  - Before this release, DataLab's validation process was exclusively done from the programmer's point of view, by writing unit tests and integration tests, thus ensuring that the code was working as expected (i.e. that no exception was raised and that the behavior was correct)
  - With this release, a new validation process has been introduced, from the user's point of view, by adding new validation functions (marked with the `@pytest.mark.validation` decorator) in the test suite
  - A new "Validation" section in the documentation explains how validation is done and contains a list of all validation functions with the statistics of the validation process (generated from the test suite)
  - The validation process is a work in progress and will be improved in future versions
- "Properties" group box:
  - Added "Scales" tab, to show and set the plot scales:
    * X, Y for signals
    * X, Y, Z (LUT range) for images
- View options:

  - New "Show first only" option in the "View" menu, to show only the first curve (or image) when multiple curves (or images) are displayed in the plot view
  - New (movable) label for FWHM computations, additional to the existing segment annotation
- I/O features:
  - Added support for reading and writing .MAT files (MATLAB format)
  - Create a new group when opening a file containing multiple signals or images (e.g. CSV file with multiple curves)
- Add support for binary images
- Signal ROI extraction: added new dialog box to manually edit the ROI lower and upper bounds after defining the ROI graphically

New **Signal** operations, processing and analysis features:

Menu	Submenu	Features
New	New signal	Exponential, pulse, polynomial, custom (manual input)
Operations		Exponential, Square root, Power
Operations	Operations with a constant	+, -, *, /
Processing	Axis Transformation	Reverse X-axis
Processing	Level Adjustment	Offset correction
Processing	Fourier analysis	Power spectrum, Phase spectrum, Magnitude spectrum, Power spectral density
Processing	Frequency filters	Low-pass, High-pass, Band-pass, Band-stop
Processing		Windowing (Hanning, Hamming, Blackman, Blackman-Harris, Nuttall, Flat-top…)
Processing	Fit	Linear fit, Sinusoidal fit, Exponential fit, CDF fit
Analysis		FWHM (Zero-crossing method), X value @ min/max, Sampling period/frequency, Dynamic parameters (ENOB, SNR, SINAD, THD, SFDR), -3dB bandwidth, Contrast

New **Image** operations, processing and analysis features:

Menu	Submenu	Features
Operations		Exponential
Operations	Intensity profiles	Profile along a segment
Operations	Operations with a constant	+, -, *, /
Processing	Level Adjustment	Normalization, Clipping, Offset correction
Processing	Fourier analysis	Power spectrum, Phase spectrum, Magnitude spectrum, Power spectral density
Processing	Thresholding	Parametric, ISODATA, Li, Mean, Minimum, Otsu, Triangle, Yen

Bug fixes:

- Fixed a performance issue due to an unnecessary refresh of the plot view when adding a new signal or image
- Fixed Issue #77 - Intensity profiles: unable to accept dialog the second time
- Fixed Issue #75 - View in a new window: curve anti-aliasing is not enabled by default
- Annotations visibility is now correctly saved and restored:
  - Before this release, when modifying the annotations visibility in the separate plot view, the visibility was not saved and restored when reopening the plot view
  - This has been fixed upstream in PlotPy (v2.3.3)

4.16 DataLab Version 0.15.1

Bug fixes:

- Fixed Issue #68 - Slow loading of even simple plots:
  - On macOS, the user experience was degraded when handling even simple plots
  - This was due to the way macOS handles the pop-up windows, e.g. when refreshing the plot view ("Creating plot items" progress bar), hence causing a very annoying flickering effect and a global slowdown of the application
  - This is now fixed by showing the progress bar only after a short delay (1s), that is when it is really needed (i.e. for long operations)
  - Thanks to @marcel-goldschen-ohm for the very thorough feedback and the help in testing the fix
- Fixed Issue #69 - Annotations should be read-only in Signal/Image View
  - Regarding the annotations, DataLab's current behavior is the following:
    * Annotations are created only when showing the signal/image in a separate window (double-click on the object, or "View" > "View in a new window")
    * When displaying the objects in either the "Signal View" or the "Image View", the annotations should be read-only (i.e. not movable, nor resizable or deletable)
  - However, some annotations were still deletable in the "Signal View" and the "Image View": this is now fixed

  - Note that the fact that annotations can't be created in the "Signal View" or the "Image View" is a limitation of the current implementation, and may be improved in future versions

4.17 DataLab Version 0.15.0

New installer for the stand-alone version on Windows:

- The stand-alone version on Windows is now distributed as an MSI installer (instead of an EXE installer)
- This avoids the false positive detection of the stand-alone version as a potential threat by some antivirus software
- The program will install files and shortcuts:
  - For current user, if the user has no administrator privileges
  - For all users, if the user has administrator privileges
  - Installation directory may be customized
- MSI installer allows to integrate DataLab's installation seemlessly in an organization's deployment system

New features and enhancements:

- Added support for large text/CSV files:
  - Files over 1 GB (and with reasonable number of lines) can now be imported as signals or images without crashing the application or even slowing it down
  - The file is read by chunks and, for signals, the data is downsampled to a reasonable number of points for visualization
  - Large files are supported when opening a file (or dragging and dropping a file in the Signal Panel) and when importing a file in the Text Import Wizard
- Auto downsampling feature:
  - Added "Auto downsampling" feature to signal visualization settings (see "Settings" dialog box)
  - This feature allows to automatically downsample the signal data for visualization when the number of points is too high and would lead to a slow rendering
  - The downsampling factor is automatically computed based on the configured maximum number of points to display
  - This feature is enabled by default and may be disabled in the signal visualization settings
- CSV format handling:
  - Improved support for CSV files with a header row (column names)
  - Added support for CSV files with empty columns
- Open/save file error handling:
  - Error messages are now more explicit when opening or saving a file fails
  - Added a link to the folder containing the file in the error message
- Added "Plugins and I/O formats" page to the Installation and Configuration Viewer (see "Help" menu)
- Reset DataLab configuration:
  - In some cases, it may be useful to reset the DataLab configuration file to its default values (e.g. when the configuration file is corrupted)
  - Added new `--reset` command line option to remove the configuration folder

  - Added new "Reset DataLab" Start Menu shortcut to the Windows installer

Bug fixes:

- Fixed Issue #64 - HDF5 browser does not show datasets with 1x1 size:
  - HDF5 datasets with a size of 1x1 were not shown in the HDF5 browser
  - Even if those datasets should not be considered as signals or images, they are now shown in the HDF5 browser (but not checkable, i.e. not importable as signals or images)

4.18 DataLab Version 0.14.2

API changes required for fixing support for multiple signals loading feature:

- Merged `open_object` and `open_objects` methods to `load_from_files` in proxy classes, main window and data panels
- For consistency's sake: merged `save_object` and `save_objects` into `save_to_files`
- To sum up, those changes lead to the following situation:
  - `load_from_files`: load a sequence of objects from multiple files
  - `save_to_files`: save a sequence of objects to multiple files (at the moment, it only supports saving a single object to a single file, but it may be extended in the future to support saving multiple objects to a single file)

Bug fixes:

- Fixed Issue #61 - Text file import wizard: application crash when importing a multiple curve text file:
  - This issue concerns a use case where the text file contains multiple curves
  - This is now fixed and an automatic test has been added to prevent regressions

4.19 DataLab Version 0.14.1

New domain name: datalab-platform.com

New features:

- Added support for colormap inversion in Image View:
  - New "Invert colormap" entry in plot context menu, image parameters, and in the default image view settings
  - This requires `PlotPy` v2.3 or later
- HDF5 Browser:
  - Added "Show array" button at the corner of the "Group" and "Attributes" tabs, to show the array in a separate window (useful for copy/pasting data to other applications, for instance)
  - Attributes: added support for more scalar data types
- Testability and maintainability:
  - DataLab's unit tests are now using pytest. This has required a lot of work for the transition, especially to readapt the tests so that they may be executed in the same process. For instance, a particular attention has been given to sandboxing the tests, so that they do not interfere with each other.
  - Added continuous integration (CI) with GitHub Actions

  - For this release, test coverage is 87%
- Text file import assistant:
  - Drastically improved the performance of the array preview when importing large text files (no more progress bar, and the preview is now displayed almost instantaneously)

Bug fixes:

- XML-RPC server was not shut down properly when closing DataLab
- Fixed test-related issues: some edge cases were hidden by the old test suite, and have been revealed by the transition to `pytest`. This has led to some bug fixes and improvements in the code.
- On Linux, when running a computation on a signal or an image, and on rare occasions, the computation was stuck as if it was running indefinitely. Even though the graphical user interface was still responsive, the computation was not progressing and the user had to cancel the operation and restart it. This was due to the start method of the separate process used for the computation (default method was "fork" on Linux). This is now fixed by using the "spawn" method instead, which is the recommended method for latest versions of Python on Linux when multithreading is involved.
- Fixed Issue #60 - `OSError: Invalid HDF5 file [...]` when trying to open an HDF5 file with an extension other than ".h5"
- Image Region of Interest (ROI) extraction: when modifying the image bounds in the confirmation dialog box, the ROI was not updated accordingly until the operation was run again
- Deprecation issues:
  - Fixed `scipy.ndimage.filters` deprecation warning
  - Fixed `numpy.fromstring` deprecation warning

4.20 DataLab Version 0.14.0

New features:

- New "Histogram" feature in "Analysis" menu:
  - Added histogram computation feature for both signals and images
  - The histogram is computed on the regions of interest (ROI) if any, or on the whole signal/image if no ROI is defined
  - Editable parameters: number of bins, lower and upper bounds
- HDF5 browser:
  - Improved tree view layout (more compact and readable)
  - Multiple files can now be opened at once, using the file selection dialog box
  - Added tabs with information below the graphical preview:
    * Group info: path, textual preview, etc.
    * Attributes info: name, value
  - Added "Show only supported data" check box: when checked, only supported data (signals and images) are shown in the tree view
  - Added "Show values" check box, to show/hide the values in the tree view
- Macro Panel:

  - Macro commands are now numbered, starting from 1, like signals and images
- Remote control API (`RemoteProxy` and `LocalProxy`):
  - `get_object_titles` method now accepts "macro" as panel name and returns the list of macro titles
  - New `run_macro`, `stop_macro` and `import_macro_from_file` methods

Bug fixes:

- Stand-alone version - Integration in Windows start menu:
  - Fixed "Uninstall" shortcut (unclickable due to a generic name)
  - Translated "Browse installation directory" and "Uninstall" shortcuts
- Fixed Issue #55 - Changing image bounds in Image View has no effect on the associated image object properties
- Fixed Issue #56 - "Test data" plugin: `AttributeError: 'NoneType' object has no attribute 'data'` when canceling "Create image with peaks"
- Fixed Issue #57 - Circle and ellipse result shapes are not transformed properly
- Curve color and style cycle:
  - Before this release, this cycle was handled by the same mechanism either for the Signal Panel or the HDF5 Browser, which was not the expected behavior
  - Now, the cycle is handled separately: the HDF5 Browser or the Text Import Wizard use always the same color and style for curves, and they don't interfere with the Signal Panel cycle

4.21 DataLab Version 0.12.0

Clarity-Enhanced Interface Update:

- The tabs used to switch between the data panels (signals and images) and the visualization components ("Curve panel" and "Image panel") have been renamed to "Signal Panel" and "Image Panel" (instead of "Signals" and "Images")
- The visualization components have been renamed to "Signal View" and "Image View" (instead of "Curve panel" and "Image panel")
- The data panel toolbar has been renamed to "Signal Toolbar" and "Image Toolbar" (instead of "Signal Processing Toolbar" and "Image Processing Toolbar")
- Ergonomics improvements: the "Signal Panel" and "Image Panel" are now displayed on the left side of the main window, and the "Signal View" and "Image View" are displayed on the right side of the main window. This reduces the distance between the list of objects (signals and images) and the associated actions (toolbars and menus), and makes the interface more intuitive and easier to use

New tour and demo feature:

- When starting DataLab for the first time, an optional tour is now shown to the user to introduce the main features of the application
- The tour can be started again at any time from the "?" menu
- Also added a new "Demo" feature to the "?" menu

New Binder environment to test DataLab online without installing anything

Documentation:

- New text tutorials are available:

  - Measuring Laser Beam Size
  - DataLab and Spyder: a perfect match
- "Getting started" section: added more explanations and links to the tutorials
- New "Contributing" section explaining how to contribute to DataLab, whether you are a developer or not
- New "Macros" section explaining how to use the macro commands feature
- Added "Copy" button to code blocks in the documentation

New features:

- New "Text file import assistant" feature:
  - This feature allows to import text files as signals or images
  - The user can define the source (clipboard or texte file)
  - Then, it is possible to define the delimiter, the number of rows to skip, the destination data type, etc.
- Added menu on the "Signal Panel" and "Image Panel" tabs corner to quickly access the most used features (e.g. "Add", "Remove", "Duplicate", etc.)
- Intensity profile extraction feature:
  - Added graphical user interface to extract intensity profiles from images, for both line and averaged profiles
  - Parameters are still directly editable by the user ("Edit profile parameters" button)
  - Parameters are now stored from one profile extraction to another
- Statistics feature:
  - Added `<y>/(y)` to the signal "Statistics" result table (in addition to the mean, median, standard deviation, etc.)
  - Added `peak-to-peak` to the signal and image "Statistics" result table
- Curve fitting feature: fit results are now stored in a dictionary in the signal metadata (instead of being stored individually in the signal metadata)
- Window state:
  - The toolbars and dock widgets state (visibility, position, etc.) are now stored in the configuration file and restored at startup (size and position were already stored and restored)
  - This implements part of Issue #30 - Save/restore main window layout

Bug fixes:

- Fixed Issue #41 - Radial profile extraction: unable to enter user-defined center coordinates
- Fixed Issue #49 - Error when trying to open a (UTF-8 BOM) text file as an image
- Fixed Issue #51 - Unexpected dimensions when adding new ROI on an image with X/Y arbitrary units (not pixels)
- Improved plot item style serialization management:
  - Before this release, the plot item style was stored in the signal/image metadata only when saving the workspace to an HDF5 file. So, when modifying the style of a signal/image from the "Parameters" button (view toolbar), the style was not kept in some cases (e.g. when duplicating the signal/image).
  - Now, the plot item style is stored in the signal/image metadata whenever the style is modified, and is restored when reloading the workspace
- Handled `ComplexWarning` cast warning when adding regions of interest (ROI) to a signal with complex data

4.22 DataLab Version 0.11.0

New features:

- Signals and images may now be reordered in the tree view:
  - Using the new "Move up" and "Move down" actions in the "Edit" menu (or using the corresponding toolbar buttons):
  - This fixes Issue #22 - Add "move up/down" actions in "Edit" menu, for signals/images and groups
- Signals and images may also be reordered using drag and drop:
  - Signals and images can be dragged and dropped inside their own panel to change their order
  - Groups can also be dragged and dropped inside their panel
  - The feature also supports multi-selection (using the standard Ctrl and Shift modifiers), so that multiple signals/images/groups can be moved at once, not necessarily with contiguous positions
  - This fixes Issue #17 - Add Drag and Drop feature to Signals/Images tree views
- New 1D interpolation features:
  - Added "Interpolation" feature to signal panel's "Processing" menu
  - Methods available: linear, spline, quadratic, cubic, barycentric and PCHIP
  - Thanks to @marcel-goldschen-ohm for the contribution to spline interpolation
  - This fixes Issue #20 - Add 1D interpolation features
- New 1D resampling feature:
  - Added "Resampling" feature to signal panel's "Processing" menu
  - Same interpolation methods as for the "Interpolation" feature
  - Possibility to specify the resampling step or the number of points
  - This fixes Issue #21 - Add 1D resampling feature
- New 1D convolution feature:
  - Added "Convolution" feature to signal panel's "Operation" menu
  - This fixes Issue #23 - Add 1D convolution feature
- New 1D detrending feature:
  - Added "Detrending" feature to signal panel's "Processing" menu
  - Methods available: linear or constant
  - This fixes Issue #24 - Add 1D detrending feature
- 2D analysis results:
  - Before this release, 2D analysis results such as contours, blobs, etc. were stored in image metadata dictionary as coordinates (x0, y0, x1, y1, ...) even for circles and ellipses (i.e. the coordinates of the bounding rectangles).
  - For convenience, the circle and ellipse coordinates are now stored in image metadata dictionary as (x0, y0, radius) and (x0, y0, a, b, theta) respectively.
  - These results are also shown as such in the "Results" dialog box (either at the end of the computing process or when clicking on the "Show results" button).

  - This fixes Issue #32 - Contour detection: show circle `(x, y, r)` and ellipse `(x, y, a, b, theta)` instead of `(x0, y0, x1, x1, ...)`

- 1D and 2D analysis results:
  - Additionnaly to the previous enhancement, more analysis results are now shown in the "Results" dialog box
  - This concerns both 1D (FHWM...) and 2D analysis results (contours, blobs...):
    * Segment results now also show length (L) and center coordinates (Xc, Yc)
    * Circle and ellipse results now also show area (A)
- Added "Plot results" entry in "Analysis" menu:
  - This feature allows to plot analysis results (1D or 2D)
  - It creates a new signal with X and Y axes corresponding to user-defined parameters (e.g. X = indices and Y = radius for circle results)
- Increased default width of the object selection dialog box:
  - The object selection dialog box is now wider by default, so that the full signal/image/group titles may be more easily readable
- Delete metadata feature:
  - Before this release, the feature was deleting all metadata, including the Regions Of Interest (ROI) metadata, if any.
  - Now a confirmation dialog box is shown to the user before deleting all metadata if the signal/image has ROI metadata: this allows to keep the ROI metadata if needed.
- Image profile extraction feature: added support for masked images (when defining regions of interest, the areas outside the ROIs are masked, and the profile is extracted only on the unmasked areas, or averaged on the unmasked areas in the case of average profile extraction)
- Curve style: added "Reset curve styles" in "View" menu. This feature allows to reset the curve style cycle to its initial state.
- Plugin base classe `PluginBase`:
  - Added `edit_new_signal_parameters` method for showing a dialog box to edit parameters for a new signal
  - Added `edit_new_image_parameters` method for showing a dialog box to edit parameters for a new image (updated the *datalab_testdata.py* plugin accordingly)
- Signal and image computations API (`datalab.computations`):
  - Added wrappers for signal and image 1 -> 1 computations
  - These wrappers aim at simplifying the creation of a basic computation function operating on DataLab's native objects (`SignalObj` and `ImageObj`) from a function operating on NumPy arrays
  - This simplifies DataLab's internals and makes it easier to create new computing features inside plugins
  - See the *datalab_custom_func.py* example plugin for a practical use case
- Added "Radial profile extraction" feature to image panel's "Operation" menu:
  - This feature allows to extract a radially averaged profile from an image
  - The profile is extracted around a user-defined center (x0, y0)
  - The center may also be computed (centroid or image center)

- Automated test suite:
  - Since version 0.10, DataLab's proxy object has a `toggle_auto_refresh` method to toggle the "Auto-refresh" feature. This feature may be useful to improve performance during the execution of test scripts
  - Test scenarios on signals and images are now using this feature to improve performance
- Signal and image metadata:
  - Added "source" entry to the metadata dictionary, to store the source file path when importing a signal or an image from a file
  - This field is kept while processing the signal/image, in order to keep track of the source file path

Documentation:

- New Tutorial section in the documentation:
  - This section provides a set of tutorials to learn how to use DataLab
  - The following video tutorials are available:
    - Quick demo
    - Adding your own features
  - The following text tutorials are available:
    - Processing a spectrum
    - Detecting blobs on an image
    - Measuring Fabry-Perot fringes
    - Prototyping a custom processing pipeline
- New API section in the documentation:
  - This section explains how to use DataLab as a Python library, by covering the following topics:
    - How to use DataLab algorithms on NumPy arrays
    - How to use DataLab computation features on DataLab objects (signals and images)
    - How to use DataLab I/O features
    - How to use proxy objects to control DataLab remotely
  - This section also provides a complete API reference for DataLab objects and features
  - This fixes Issue #19 - Add API documentation (data model, functions on arrays or signal/image objects, ...)

Bug fixes:

- Fixed Issue #29 - Polynomial fit error: `QDialog [...] argument 1 has an unexpected type 'SignalProcessor'`
- Image ROI extraction feature:
  - Before this release, when extracting a single circular ROI from an image with the "Extract all ROIs into a single image object" option enabled, the result was a single image without the ROI mask (the ROI mask was only available when extracting ROI with the option disabled)
  - This was leading to an unexpected behavior, because one could interpret the result (a square image without the ROI mask) as the result of a single rectangular ROI
  - Now, when extracting a single circular ROI from an image with the "Extract all ROIs into a single image object" option enabled, the result is a single image with the ROI mask (as if the option was disabled)

  - This fixes Issue #31 - Single circular ROI extraction: automatically switch to `compute_extract_roi` function

- Analysis on circular ROI:
  - Before this release, when running computations on a circular ROI, the results were unexpected in terms of coordinates (results seemed to be computed in a region located above the actual ROI).
  - This was due to a regression introduced in an earlier release.
  - Now, when defining a circular ROI and running computations on it, the results are computed on the actual ROI
  - This fixes Issue #33 - Analysis on circular ROI: unexpected results
- Contour detection on ROI:
  - Before this release, when running contour detection on a ROI, some contours were detected outside the ROI (it may be due to a limitation of the scikit-image `find_contours` function).
  - Now, thanks a workaround, the erroneous contours are filtered out.
  - A new test module `datalab.tests.features.images.contour_fabryperot_app` has been added to test the contour detection feature on a Fabry-Perot image (thanks to @emarin2642 for the contribution)
  - This fixes Issue #34 - Contour detection: unexpected results outside ROI
- Analysis result merging:
  - Before this release, when doing a `1->N` computation (sum, average, product) on a group of signals/images, the analysis results associated to each signal/image were merged into a single result, but only the type of result present in the first signal/image was kept.
  - Now, the analysis results associated to each signal/image are merged into a single result, whatever the type of result is.
- Fixed Issue #36 - "Delete all" action enable state is sometimes not refreshed
- Image X/Y swap: when swapping X and Y axes, the regions of interest (ROI) were not removed and not swapped either (ROI are now removed, until we implement the swap feature, if requested)
- "Properties" group box: the "Apply" button was enabled by default, even when no property was modified, which was confusing for the user (the "Apply" button is now disabled by default, and is enabled only when a property is modified)
- Fixed proxy `get_object` method when there is no object to return (`None` is returned instead of an exception)
- Fixed `IndexError: list index out of range` when performing some operations or computations on groups of signals/images (e.g. "ROI extraction", "Peak detection", "Resize", etc.)
- Drag and drop from a file manager: filenames are now sorted alphabetically

4.23 DataLab Version 0.10.1

Note: V0.10.0 was almost immediately replaced by V0.10.1 due to a last minute bug fix

New features:

- Features common to signals and images:
  - Added "Real part" and "Imaginary part" features to "Operation" menu
  - Added "Convert data type" feature to "Operation" menu

- Features added following user requests (12/18/2023 meetup @ CEA):
  - Curve and image styles are now saved in the HDF5 file:
    - Curve style covers the following properties: color, line style, line width, marker style, marker size, marker edge color, marker face color, etc.
    - Image style covers the following properties: colormap, interpolation, etc.
    - Those properties were already persistent during the working session, but were lost when saving and reloading the HDF5 file
    - Now, those properties are saved in the HDF5 file and are restored when reloading the HDF5 file
  - New profile extraction features for images:
    - Added "Line profile" to "Operations" menu, to extract a profile from an image along a row or a column
    - Added "Average profile" to "Operations" menu, to extract the average profile on a rectangular area of an image, along a row or a column
  - Image LUT range (contrast/brightness settings) is now saved in the HDF5 file:
    - As for curve and image styles, the LUT range was already persistent during the working session, but was lost when saving and reloading the HDF5 file
    - Now, the LUT range is saved in the HDF5 file and is restored when reloading it
  - Added "Auto-refresh" and "Refresh manually" actions in "View" menu (and main toolbar):
    - When "Auto-refresh" is enabled (default), the plot view is automatically refreshed when a signal/image is modified, added or removed. Even though the refresh is optimized, this may lead to performance issues when working with large datasets.
    - When disabled, the plot view is not automatically refreshed. The user must manually refresh the plot view by clicking on the "Refresh manually" button in the main toolbar or by pressing the standard refresh key (e.g. "F5").
  - Added `toggle_auto_refresh` method to DataLab proxy object:
    - This method allows to toggle the "Auto-refresh" feature from a macro-command, a plugin or a remote control client.
    - A context manager `context_no_refresh` is also available to temporarily disable the "Auto-refresh" feature from a macro-command, a plugin or a remote control client. Typical usage:

      ```
      with proxy.context_no_refresh():
          # Do something without refreshing the plot view
          proxy.compute_fft() # (...)
      ```

  - Improved curve readability:
    - Until this release, the curve style was automatically set by cycling through **PlotPy** predefined styles
    - However, some styles are not suitable for curve readability (e.g. "cyan" and "yellow" colors are not readable on a white background, especially when combined with a "dashed" line style)
    - This release introduces a new curve style management with colors which are distinguishable and accessible, even to color vision deficiency people
- Added "Curve anti-aliasing" feature to "View" menu (and toolbar):
  - This feature allows to enable/disable curve anti-aliasing (default: enabled)
  - When enabled, the curve rendering is smoother but may lead to performance issues when working with large datasets (that's why it can be disabled)

- Added `toggle_show_titles` method to DataLab proxy object. This method allows to toggle the "Show graphical object titles" feature from a macro-command, a plugin or a remote control client.
- Remote client is now checking the server version and shows a warning message if the server version may not be fully compatible with the client version.

Bug fixes:

- Image contour detection feature ("Analysis" menu):
  - The contour detection feature was not taking into account the "shape" parameter (circle, ellipse, polygon) when computing the contours. The parameter was stored but really used only when calling the feature a second time.
  - This unintentional behavior led to an `AssertionError` when choosing "polygon" as the contour shape and trying to compute the contours for the first time.
  - This is now fixed (see Issue #9 - Image contour detection: `AssertionError` when choosing "polygon" as the contour shape)
- Keyboard shortcuts:
  - The keyboard shortcuts for "New", "Open", "Save", "Duplicate", "Remove", "Delete all" and "Refresh manually" actions were not working properly.
  - Those shortcuts were specific to each signal/image panel, and were working only when the panel on which the shortcut was pressed for the first time was active (when activated from another panel, the shortcut was not working and a warning message was displayed in the console, e.g. `QAction::event: Ambiguous shortcut overload: Ctrl+C`)
  - Besides, the shortcuts were not working at startup (when no panel had focus).
  - This is now fixed: the shortcuts are now working whatever the active panel is, and even at startup (see Issue #10 - Keyboard shortcuts not working properly: `QAction::event: Ambiguous shortcut overload: Ctrl+C`)
- "Show graphical object titles" and "Auto-refresh" actions were not working properly:
  - The "Show graphical object titles" and "Auto-refresh" actions were only working on the active signal/image panel, and not on all panels.
  - This is now fixed (see Issue #11 - "Show graphical object titles" and "Auto-refresh" actions were working only on current signal/image panel)
- Fixed Issue #14 - Saving/Reopening HDF5 project without cleaning-up leads to `ValueError`
- Fixed Issue #15 - MacOS: 1. `pip install cdl` error - 2. Missing menus:
  - Part 1: `pip install cdl` error on MacOS was actually an issue from **PlotPy** (see this issue), and has been fixed in PlotPy v2.0.3 with an additional compilation flag indicating to use C++11 standard
  - Part 2: Missing menus on MacOS was due to a PyQt/MacOS bug regarding dynamic menus
- HDF5 file format: when importing an HDF5 dataset as a signal or an image, the dataset attributes were systematically copied to signal/image metadata: we now only copy the attributes which match standard data types (integers, floats, strings) to avoid errors when serializing/deserializing the signal/image object
- Installation/configuration viewer: improved readability (removed syntax highlighting)
- PyInstaller specification file: added missing `skimage` data files manually in order to continue supporting Python 3.8 (see Issue #12 - Stand-alone version on Windows 7: missing `api-ms-win-core-path-l1-1-0.dll`)
- Fixed Issue #13 - ArchLinux: `qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even though it was found`

4.24 DataLab Version 0.9.2

Bug fixes:

- Region of interest (ROI) extraction feature for images:
  - ROI extraction was not working properly when the "Extract all ROIs into a single image object" option was enabled if there was only one defined ROI. The result was an image positioned at the origin (0, 0) instead of the expected position (x0, y0) and the ROI rectangle itself was not removed as expected. This is now fixed (see Issue #6 - 'Extract multiple ROI' feature: unexpected result for a single ROI)
  - ROI rectangles with negative coordinates were not properly handled: ROI extraction was raising a `ValueError` exception, and the image mask was not displayed properly. This is now fixed (see Issue #7 - Image ROI extraction: `ValueError: zero-size array to reduction operation minimum which has no identity`)
  - ROI extraction was not taking into account the pixel size (dx, dy) and the origin (x0, y0) of the image. This is now fixed (see Issue #8 - Image ROI extraction: take into account pixel size)
- Macro-command console is now read-only:
  - The macro-command panel Python console is currently not supporting standard input stream (`stdin`) and this is intended (at least for now)
  - Set Python console read-only to avoid confusion

4.25 DataLab Version 0.9.1

Bug fixes:

- French translation is not available on Windows/Stand alone version:
  - Locale was not properly detected on Windows for stand-alone version (frozen with `pyinstaller`) due to an issue with `locale.getlocale()` (function returning `None` instead of the expected locale on frozen applications)
  - This is ultimately a `pyinstaller` issue, but a workaround has been implemented in `guidata` V3.2.2 (see guidata issue #68 - Windows: gettext translation is not working on frozen applications)
  - Issue #2 - French translation is not available on Windows Stand alone version
- Saving image to JPEG2000 fails for non integer data:
  - JPEG2000 encoder does not support non integer data or signed integer data
  - Before, DataLab was showing an error message when trying to save incompatible data to JPEG2000: this was not a consistent behavior with other standard image formats (e.g. PNG, JPG, etc.) for which DataLab was automatically converting data to the appropriate format (8-bit unsigned integer)
  - Current behavior is now consistent with other standard image formats: when saving to JPEG2000, DataLab automatically converts data to 8-bit unsigned integer or 16-bit unsigned integer (depending on the original data type)
  - Issue #3 - Save image to JPEG2000: 'OSError: encoder error -2 when writing image file'
- Windows stand-alone version shortcuts not showing in current user start menu:
  - When installing DataLab on Windows from a non-administrator account, the shortcuts were not showing in the current user start menu but in the administrator start menu instead (due to the elevated privileges of the installer and the fact that the installer does not support installing shortcuts for all users)

  - Now, the installer *does not* ask for elevated privileges anymore, and shortcuts are installed in the current user start menu (this also means that the current user must have write access to the installation directory)
  - In future releases, the installer will support installing shortcuts for all users if there is a demand for it (see Issue #5)
  - Issue #4 - Windows: stand-alone version shortcuts not showing in current user start menu
- Installation and configuration window for stand-alone version:
  - Do not show ambiguous error message 'Invalid dependencies' anymore
  - Dependencies are supposed to be checked when building the stand-alone version
- Added PDF documentation to stand-alone version:
  - The PDF documentation was missing in previous release
  - Now, the PDF documentation (in English and French) is included in the stand-alone version

4.26 DataLab Version 0.9.0

New dependencies:

- DataLab is now powered by PlotPyStack:
  - PythonQwt
  - guidata
  - PlotPy
- opencv-python (algorithms for image processing)

New reference platform:

- DataLab is validated on Windows 11 with Python 3.11 and PyQt 5.15
- DataLab is also compatible with other OS (Linux, MacOS) and other Python-Qt bindings and versions (Python 3.8-3.12, PyQt6, PySide6)

New features:

- DataLab is a platform:
  - Added support for plugins
    * Custom processing features available in the "Plugins" menu
    * Custom I/O features: new file formats can be added to the standard I/O features for signals and images
    * Custom HDF5 features: new HDF5 file formats can be added to the standard HDF5 import feature
    * More features to come…
  - Added remote control feature: DataLab can be controlled remotely via a TCP/IP connection (see Remote control)
  - Added macro commands: DataLab can be controlled via a macro file (see Macro commands)
- General features:
  - Added settings dialog box (see "Settings" entry in "File" menu):
    * General settings
    * Visualization settings

    * Processing settings
    * Etc.
  - New default layout: signal/image panels are on the right side of the main window, visualization panels are on the left side with a vertical toolbar
- Signal/Image features:
  - Added process isolation: each signal/image is processed in a separate process, so that DataLab does not freeze anymore when processing large signals/images
  - Added support for groups: signals and images can be grouped together, and operations can be applied to all objects in a group, or between groups
  - Added warning and error dialogs with detailed traceback links to the source code (warnings may be optionally ignored)
  - Drastically improved performance when selecting objects
  - Optimized performance when showing large images
  - Added support for dropping files on signal/image panel
  - Added "Analysis parameters" group box to show last result input parameters
  - Added "Copy titles to clipboard" feature in "Edit" menu
  - For every single processing feature (operation, processing and analysis menus), the entered parameters (dialog boxes) are stored in cache to be used as defaults the next time the feature is used
- Signal processing:
  - Added support for optional FFT shift (see Settings dialog box)
- Image processing:
  - Added pixel binning operation (X/Y binning factors, operation: sum, mean…)
  - Added "Distribute on a grid" and "Reset image positions" in operation menu
  - Added Butterworth filter
  - Added exposure processing features:
    * Gamma correction
    * Logarithmic correction
    * Sigmoïd correction
  - Added restoration processing features:
    * Total variation denoising filter (TV Chambolle)
    * Bilateral filter (denoising)
    * Wavelet denoising filter
    * White Top-Hat denoising filter
  - Added morphological transforms (disk footprint):
    * White Top-Hat
    * Black Top-Hat
    * Erosion
    * Dilation

    - Opening
    - Closing
  - Added edge detection features:
    - Roberts filter
    - Prewitt filter (vertical, horizontal, both)
    - Sobel filter (vertical, horizontal, both)
    - Scharr filter (vertical, horizontal, both)
    - Farid filter (vertical, horizontal, both)
    - Laplace filter
    - Canny filter
  - Contour detection: added support for polygonal contours (in addition to circle and ellipse contours)
  - Added circle Hough transform (circle detection)
  - Added image intensity levels rescaling
  - Added histogram equalization
  - Added adaptative histogram equalization
  - Added blob detection methods:
    - Difference of Gaussian
    - Determinant of Hessian method
    - Laplacian of Gaussian
    - Blob detection using OpenCV
  - Result shapes and annotations are now transformed (instead of removed) when executing one of the following operations:
    - Rotation (arbitrary angle, +90°, -90°)
    - Symetry (vertical/horizontal)
  - Added support for optional FFT shift (see Settings dialog box)
- Console: added configurable external editor (default: VSCode) to follow the traceback links to the source code

Note: DataLab was created by CODRA/Pierre Raybaut in 2023. It is developed and maintained by DataLab Platform Developers.

Note: This project (DataLab Platform) should not be confused with the datalab-org project, which is a separate and unrelated initiative focused on materials science databases and computational tools.

PYTHON MODULE INDEX

d

datalab.control.proxy, 244
datalab.gui, 258
datalab.gui.actionhandler, 291
datalab.gui.docks, 307
datalab.gui.h5io, 308
datalab.gui.main, 258
datalab.gui.objectview, 297
datalab.gui.panel, 266
datalab.gui.panel.base, 266
datalab.gui.panel.image, 282
datalab.gui.panel.macro, 287
datalab.gui.panel.signal, 279
datalab.gui.plothandler, 300
datalab.gui.processor, 306
datalab.gui.roieditor, 304
datalab.plugins, 204