

DataLab

Release 1.4.1

Nov 15, 2026

CONTENTS

1 Getting started 3
- 1.1 Installation . . . 4
- 1.2 Offline Installation . . . 11
- 1.3 Use cases, main features and key strengths . . . 17
- 1.4 The DataLab Platform . . . 19
- 1.5 Key features . . . 23
- 1.6 Tutorials . . . 26

2 Use cases 113
- 2.1 DataLab for spectroscopy . . . 113
- 2.2 DataLab for photonics and lasers . . . 114
- 2.3 DataLab for non-destructive testing . . . 115
- 2.4 DataLab for scientific camera characterization . . . 117
- 2.5 DataLab for transient and pulsed signals . . . 119
- 2.6 DataLab for instrument control and data acquisition . . . 121

3 Features 123
- 3.1 Overview & Common features . . . 123
- 3.2 Signal processing . . . 158
- 3.3 Image processing . . . 203
- 3.4 Validation . . . 249
- 3.5 Advanced features . . . 258

4 Contributing 443
- 4.1 Share your ideas and experiences . . . 443
- 4.2 Share your scientific/technical knowledge . . . 443
- 4.3 Contribute to new features . . . 443
- 4.4 Develop new features . . . 444

5 Release notes 463
- 5.1 Version 1.4 . . . 463
- 5.2 Version 1.3 . . . 483
- 5.3 Version 1.2 . . . 488
- 5.4 Version 1.1 . . . 493
- 5.5 Version 1.0 . . . 495
- 5.6 Version 0.20 . . . 510
- 5.7 Version 0.19 . . . 512
- 5.8 Version 0.18 . . . 514
- 5.9 Version 0.17 . . . 517
- 5.10 Version 0.16 . . . 520

5.11 Version 0.15 524
5.12 Version 0.14 526
5.13 Version 0.12 528
5.14 Version 0.11 530
5.15 Version 0.10 534
5.16 Version 0.9 536

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 can be extended with plugins: browse the DataLab plugin catalog, or create your own plugin (see From an idea to the plugin catalog).

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

cea	CEA, the French Alternative Energies and Atomic Energy Commission, is the major investor in DataLab, and is the main contributor to the project.
CODRA	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 (optional) 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). Its graphical user interface is built on SigimaX, the Qt application framework of the DataLab Platform.

The desktop application itself is built on SigimaX, the Qt application framework of the DataLab Platform: main window skeleton, configuration system, dockable plot widgets, HDF5 workspace and browser, log viewer and scientific dialogs (see 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 seamlessly 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.

> **Note**
>
> DataLab is the desktop application at the heart of a broader family, the DataLab Platform, which also includes a browser-native edition (DataLab-Web) that runs without any installation. See The DataLab Platform to learn about the editions and choose the one that fits your needs.

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)
- SigimaX for the Qt application framework the desktop application is built upon (part of the DataLab Platform)
- DataLab-Capsule for the provenance of the workspace and the RO-Crate `.dlcapsule` workspace capsules (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 (optional) 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.
>
> For installation on systems without internet access, see the offline installation guide.

1.1.1 How to install

DataLab is available in several forms:

- As a Conda package.
- GNU/Linux macOS As a NixOS package, for users of the NixOS ecosystem.
- 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 `develop` 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
```

NixOS package

GNU/Linux macOS

DataLab is packaged for NixOS and distributed through the NGI Forge, the software distribution system for projects funded by the Next Generation Internet (NGI) initiative. This packaging was contributed by the Nix@NGI team, an NLnet partner, as part of DataLab's funding through the NGI0 Commons Fund.

On any system with Nix installed, enter the Nix shell that provides DataLab (the ready-to-copy command is available on the NGI Forge page), then start the desktop application:

```
$ datalab
```

Or, for an automated demonstration of some of DataLab's features:

```
$ datalab-demo
```

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
```

The optional `opencv` extra installs OpenCV, which is required only by the OpenCV blob detection feature. To install it along with Qt:

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

> **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!

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).

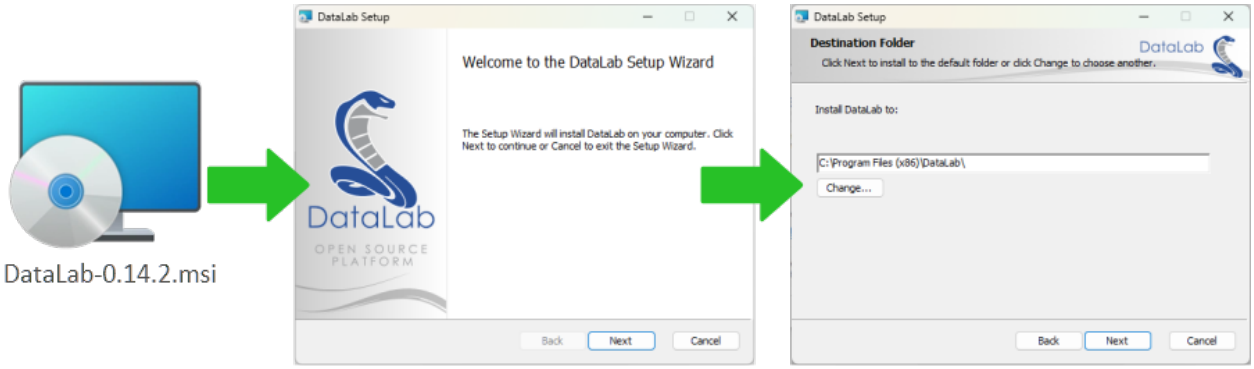


Fig. 1: DataLab all-in-one installer for Windows

> **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.

> **Note**
>
> DataLab-WinPython includes Spyder, a powerful IDE for scientific programming in Python, as well as Jupyter Notebook for interactive computing.

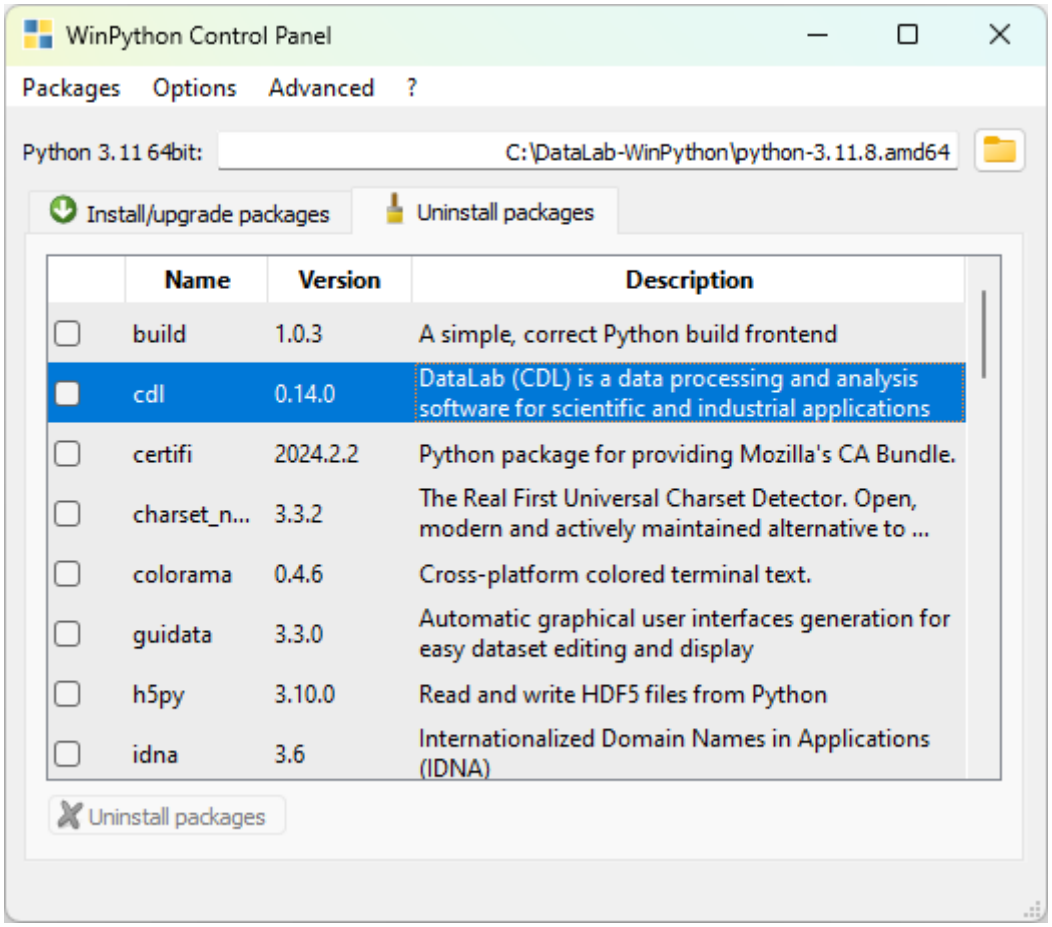


Fig. 3: DataLab-WinPython Control Panel

> **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_platform-X.Y.Z-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_platform-X.Y.Z.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_platform-X.Y.Z-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 `develop` branch of the Git repository.

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

```
$ pip install "datalab-platform[qt] @ git+https://github.com/DataLab-Platform/DataLab.
↪git@develop"
```

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@develop
```

> **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	>=3.10, <4	Python programming language
guidata	>= 3.17.3	Automatic GUI generation for easy dataset editing and display
PlotPy	>= 2.12.1	Curve and image plotting tools for Python/Qt applications
Sigima	>= 1.4.1	Scientific computing engine for 1D signals and 2D images, part of the DataLab open-source platform.
SigimaX	>= 1.1.1	Reusable GUI components and application utilities, part of the DataLab platform for signal and image processing
datalab-capsule	>= 0.2.0, < 0.3	Exchange and verify scientific workspace provenance as open RO-Crate capsules (library and CLI, no DataLab required)
NumPy	>= 1.22, < 2.6	Fundamental package for array computing in Python
SciPy	>= 1.10.1, < 1.19	Fundamental algorithms for scientific computing in Python
scikit-image	>= 0.19.2, < 0.27	Image processing in Python
pandas	>= 1.4, < 3.1	Powerful data structures for data analysis, time series, and statistics
PyWavelets	>= 1.2, < 2.0	PyWavelets, wavelet transform module
psutil	>= 5.8	Cross-platform lib for process and system monitoring.
packaging	>= 21.3	Core utilities for Python packages
pip	>= 22.2	The PyPA recommended tool for installing Python packages.
jedi	>= 0.18.2	An autocompletion tool for Python that can be used for text editors.
fastapi	>= 0.110.0	FastAPI framework, high performance, easy to learn, fast to code, ready for production
uvicorn[standard]	>= 0.27.0	
pydantic	>= 2.0	Data validation using Python type hints

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
sphinxext-opengraph		Sphinx Extension to enable OGP support
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
sphinxcontrib-svg2pdfconverter[CairoSVG]		

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.
httpx		The next generation HTTP client.
pyyaml		YAML parser and emitter for Python

The optional `opencv` extra (`opencv-python-headless`), not listed above, is required only by the OpenCV blob detection feature (see Package manager pip).

> **Note**
>
> Python 3.11 and PyQt5 are the reference for production release

1.2 Offline Installation

This guide explains how to install DataLab on a system without internet access.

> **Note**
>
> This page focuses on offline installation using pip (Python package manager). For other offline installation methods, see the Other offline installation methods section below.

1.2.1 Offline installation via pip

GNU/Linux Windows macOS

This method works on any operating system with Python 3.10 to 3.14. It assumes you have an online machine available to download packages, and a separate offline machine where DataLab will be installed. The commands below download wheels for the online machine's platform: use an online machine with the same operating system, architecture and compatible platform ABI as the offline machine.

1.2.2 Requirements

Offline machine

- GNU/Linux, Windows, or macOS
- Python 3.10 to 3.14 installed with the same minor version (3.x) as the online machine
- The same operating system and architecture, with a compatible platform ABI
- No assumptions regarding installed packages

> **Important**
>
> Version compatibility is critical:
>
> - ✓ Allowed: Python 3.10.2 ↔ Python 3.10.11 (same minor version)
> - × Not allowed: Python 3.10.x ↔ Python 3.11.x (different minor versions)
>
> SciPy, NumPy, and scikit-image are highly dependent on the Python minor version. Both machines must use the same minor Python version for reliable offline installation.

To check your Python version, run:

```
python --version
```

Online machine

- The same operating system and architecture as the offline machine, with a compatible platform ABI
- Python with the same minor version as the offline machine
- `pip` installed

1.2.3 Preparing packages (online machine)

Download all required Wheel packages for offline installation. `pip download` selects wheels compatible with the machine running it; a matching Python version alone is not enough for a different platform.

Create a folder for packages

Windows

```
mkdir C:\datalab-offline
cd C:\datalab-offline
```

GNU/Linux / macOS

```
mkdir ~/datalab-offline
cd ~/datalab-offline
```

Download DataLab and dependencies

Windows

```
pip download "datalab-platform[qt]" --dest C:\datalab-offline
```

GNU/Linux / macOS

```
pip download "datalab-platform[qt]" --dest ~/datalab-offline
```

This command downloads:

- `datalab-platform`
- `sigima`
- `sigimax`
- `datalab-capsule`
- `plotpy`
- `guidata`
- `pythonqwt`
- `pyqt5` (Qt bindings, installed by the `qt` extra)
- `numpy`, `scipy`, `scikit-image`, and other dependencies

> **Note**
>
> The OpenCV blob detection feature requires the optional `opencv` extra: to use it, replace `datalab-platform[qt]` by `datalab-platform[qt,opencv]` in the download and installation commands.

Expected folder structure

After downloading, your folder should contain files similar to:

```
datalab-offline/
├─ datalab_platform-1.x.x-py3-none-any.whl
├─ sigima-1.x.x-py3-none-any.whl
├─ sigimax-1.x.x-py3-none-any.whl
├─ datalab_capsule-0.x.x-py3-none-any.whl
├─ plotpy-x.x.x-py3-none-any.whl
├─ guidata-3.x.x-py3-none-any.whl
├─ pythonqwt-x.x.x-py3-none-any.whl
├─ pyqt5-5.x.x-cp38-abi3-[platform].whl
├─ numpy-...-cp310-cp310-[platform].whl
├─ scipy-...-cp310-cp310-[platform].whl
├─ scikit_image-...-cp310-cp310-[platform].whl
└─ ...
```

> **Note**
>
> The `[platform]` suffix in the wheel filenames will vary depending on your system: `win_amd64` (Windows), `manylinux` (Linux), `macosx` (macOS).

Transfer to offline machine

Transfer the `datalab-offline` folder to your offline machine using:

- USB drive
- Network share
- Any other approved method in your organization

1.2.4 Offline installation

On the offline machine, navigate to the folder and install DataLab.

Navigate to the folder

Windows

```
cd C:\datalab-offline
```

GNU/Linux / macOS

```
cd ~/datalab-offline
```

Install DataLab offline

```
python -m pip install --no-index --find-links . "datalab-platform[qt]"
```

The `--no-index` option tells pip not to connect to the internet, and `--find-links .` tells pip to look for packages in the current directory.

1.2.5 Verification

To verify that DataLab is installed correctly, run:

```
python -m datalab.app
```

This should launch the DataLab application.

1.2.6 Optional: Virtual environment

It is recommended to install DataLab in a virtual environment to avoid conflicts with other Python packages.

Create and activate a virtual environment

Windows

```
python -m venv C:\venvs\datalab
C:\venvs\datalab\Scripts\activate
```

GNU/Linux / macOS

```
python -m venv ~/venvs/datalab
source ~/venvs/datalab/bin/activate
```

Install DataLab in the virtual environment

Windows

```
python -m pip install --no-index --find-links C:\datalab-offline "datalab-platform[qt]
↪"
```

GNU/Linux / macOS

```
python -m pip install --no-index --find-links ~/datalab-offline "datalab-platform[qt]"
```

Launch DataLab

```
python -m datalab.app
```

1.2.7 WinPython for offline usage

Windows

WinPython is particularly well-suited for offline installations as it provides a portable, self-contained Python environment.

Prepare on online machine

1. Download WinPython with the same minor Python version as your target system
2. Install WinPython
3. Open the WinPython Command Prompt
4. Download packages:

```
pip download "datalab-platform[qt]" --dest C:\datalab-offline
```

Install on offline machine

1. Transfer WinPython and the `datalab-offline` folder to the offline machine
2. Open the WinPython Command Prompt
3. Install DataLab:

```
cd C:\datalab-offline
python -m pip install --no-index --find-links . "datalab-platform[qt]"
```

Benefits of WinPython

- Portable: No installation required, can run from USB drive
- Self-contained: All dependencies included
- Ideal for restricted environments: No admin rights needed
- Multiple versions: Can have multiple Python versions side by side

1.2.8 Quick summary

The complete workflow in four steps:

1. On online machine: Download packages

   ```
   pip download "datalab-platform[qt]" --dest /path/to/datalab-offline
   ```

2. Transfer: Copy the `datalab-offline` folder to the offline machine
3. On offline machine: Install offline

   ```
   python -m pip install --no-index --find-links /path/to/datalab-offline "datalab-
   ↪platform[qt]"
   ```

4. Verify: Launch DataLab

   ```
   python -m datalab.app
   ```

1.2.9 Other offline installation methods

While this guide focuses on pip-based offline installation, DataLab can also be installed offline using other methods:

Stand-alone installer (Windows)

Windows Recommended for offline installation

The all-in-one installer is the simplest way to install DataLab offline on Windows. It does not require any Python distribution or internet connection on the target machine.

Advantages:

- No Python installation required
- Single executable file
- Self-contained with all dependencies

- No internet connection needed during installation
- Works on air-gapped systems

How to use:

1. On a machine with internet access, download the installer from the DataLab Releases page
2. Transfer the installer executable to the offline machine (USB drive, network share, etc.)
3. Run the installer on the offline machine

For more information, see All-in-one installer.

WinPython distribution (Windows)

Windows

DataLab-WinPython is a portable Python distribution that includes DataLab and all its dependencies.

Advantages:

- Portable (can run from USB drive)
- No installation required
- Includes full Python environment
- Can be extended with additional packages
- Ideal for restricted environments

How to use:

1. Download DataLab-WinPython from the releases page
2. Transfer to the offline machine
3. Extract and run

For more information, see Python distribution.

Conda offline installation

GNU/Linux Windows macOS

Conda packages can also be installed offline using a similar approach to pip:

1. On online machine: Download the conda package and dependencies:

   ```
   conda create -n datalab-offline --download-only --json conda-forge::datalab
   ```

   Or use `conda list --explicit` to create a specification file.
2. Transfer the downloaded packages to the offline machine
3. On offline machine: Install from the local packages:

   ```
   conda create -n datalab --offline -c file:///path/to/packages datalab
   ```

For more information about offline conda installation, see the Conda documentation.

For standard online installation, see Conda package.

See also

For standard online installation methods, see Installation.

1.3 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.

1.3.1 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 (see the DataLab plugin catalog).
-  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; an HTTP/JSON Web API is also available, see Web API). 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.

1.3.2 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)
- History panel to record and replay processing chains

- Provenance of results (verification by recomputation) and workspace capsules (`.dlcapsule`), based on DataLab-Capsule
- HDF5 workspaces and HDF5/NeXus browser
- Macro-command editor
- Remote-controllable API (XML-RPC) and HTTP/JSON Web API
- Embedded interactive Python console
- Command palette, welcome page and Applications panel
- Optional AI assistant

1.3.3 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.

  Browse the DataLab plugin catalog, or create your own plugin (see Plugins and the tutorial From an idea to the plugin catalog).

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 Validation for more information on DataLab's validation strategy.

1.4 The DataLab Platform

When DataLab started in 2023, it was a single product: a desktop application. Since then, the same scientific core has grown into a small family of complementary tools — the DataLab Platform. This page clarifies the terminology and helps you choose the edition that best fits your needs.

> **See also**
>
> The story behind this evolution — the extraction of the Sigima computation engine and the browser-native edition — is documented in the project reports and Architecture Decision Records published on the DataLab wiki.

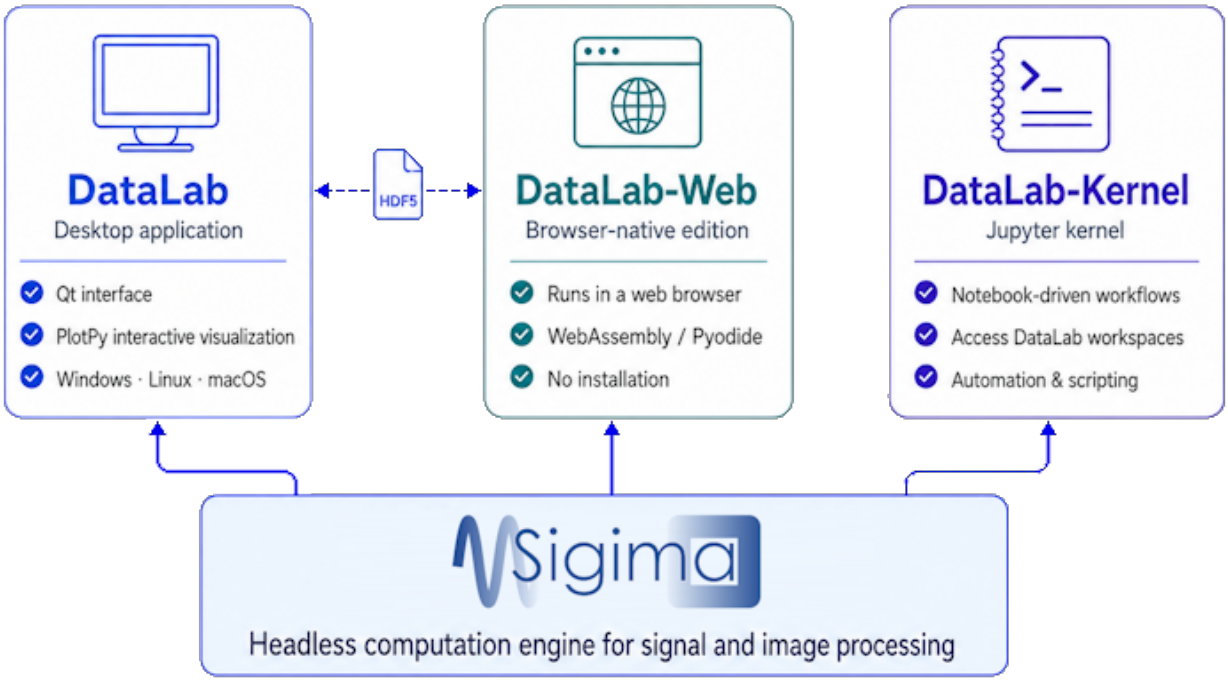


Fig. 4: The DataLab Platform brings together the desktop application, DataLab-Web and DataLab-Kernel around the Sigima computation engine. DataLab-Web can open and save HDF5 workspaces, but it does not preserve Desktop History, so round trips are not lossless.

1.4.1 A note on terminology

The word DataLab is used both for the whole family and for its historical member, the desktop application. To avoid confusion, this documentation uses the following vocabulary:

Name	Meaning
DataLab Platform	The umbrella term for the whole ecosystem (all the products listed below).
DataLab	The reference desktop application (Qt-based). Unless stated otherwise, the rest of this documentation describes this product.
DataLab-Web	The browser-native edition: a curated set of DataLab features running inside a web browser tab, with no installation. Try it at datalab-platform.com/web – project on GitHub.
DataLab-Kernel	A Jupyter kernel exposing DataLab workspaces to notebooks — either live-synchronized with a running desktop session, or fully standalone (including in Jupyter-Lite, in the browser). Documentation.
Sigima	The headless computation engine (signal and image processing) shared by all of the above. Documentation.
SigimaX	The Qt application framework on which the desktop application is built: generic main window, configuration system, dockable plot widgets, HDF5 workspace and browser, log viewer and scientific dialogs. Aimed at developers building their own Qt scientific application. Documentation.
DataLab-Capsule	The workspace provenance and capsule package (`datalab-capsule`), used by both the desktop application and DataLab-Web: it records how results were obtained in the HDF5 workspace, and packages workspaces as open RO-Crate `.dlcapsule` archives to exchange, compare and verify them. Its library and command line do not require DataLab. Project on GitHub.

> **Note**
>
> This documentation site is, first and foremost, the documentation of the desktop application. Some core concepts and operations are shared across editions, but feature availability is not identical: DataLab-Web exposes a curated subset of Sigima processing operations. Edition-specific differences are called out explicitly.

1.4.2 Members of the platform

- DataLab (desktop) — the reference application, with a native Qt graphical user interface and the full PlotPyStack interactive visualization.
- DataLab-Web — the same platform, running entirely inside your browser via WebAssembly (datalab-platform.com/web).
- DataLab-Kernel — a Jupyter kernel for notebook-driven workflows, live-synchronized with a running desktop session or fully standalone (documentation).
- Sigima — the open-source computation engine that powers every edition (documentation).
- SigimaX — the Qt application framework on which the desktop application is built, reusable to create other scientific Qt applications (documentation).
- DataLab-Capsule — the provenance of the workspace and the open RO-Crate `.dlcapsule` capsules, shared by the desktop application and DataLab-Web (project on GitHub).
- DataLab plugin catalog — the plugins of the DataLab team and of the community, checked automatically and installable in one click from the desktop application (datalab-platform.com/plugins).

1.4.3 Which one should I use?

The desktop application and DataLab-Web are two access modes to the same platform and can exchange HDF5 workspaces, but they are not fully interchangeable. DataLab-Web does not preserve Desktop History when it opens and saves an HDF5 workspace, and its processing catalog is a curated subset. A few rules of thumb:

- Use the desktop application when you work on your own machine and want the richest, fastest experience, the full set of interactive PlotPy tools, native file access, and the ability to handle very large datasets limited only by your system RAM.
- Use DataLab-Web when you cannot or do not want to install anything — for example on a shared or locked-down computer, on someone else's machine, or simply to give DataLab a quick try. Core processing runs locally in the browser without a DataLab server or account.
- Use DataLab-Kernel when your workflow is notebook-centric and you want DataLab's processing catalog available from a Jupyter notebook — for example to write reproducible analysis reports that can be replayed with or without DataLab. You can try it online in a live JupyterLite environment, without installing anything.

A common pattern is to use the desktop application on your personal workstation and DataLab-Web elsewhere, exchanging workspace data through HDF5 files. DataLab-Web does not preserve Desktop History, so round trips can lose that information.

1.4.4 DataLab vs DataLab-Web

DataLab-Web shares the computation engine (Sigima) with the desktop application and exposes a curated subset of its processing catalog. The very same Sigima code — not a port — runs on CPython (desktop) and on Pyodide (browser). For operations available in both editions, compatible versions of Sigima use the same algorithms and parameters, but numerical equivalence depends on the inputs and the numerical environment. Most operation contracts declare an exact or tolerance-based comparison of replayed results; a few declare no comparison rule. The available operations and runtime environment differ between editions.

Topic	DataLab (desktop)	DataLab-Web
Installation	Installed locally (Windows, Linux, macOS)	None — runs in the browser
Python runtime	System CPython	Pyodide (CPython compiled to WebAssembly)
Graphical interface	Qt + PlotPy	React + Plotly.js
Plotting	PlotPy (Qt), full interactive tools	Plotly.js (interactive tools partially reimplemented)
File access	Native file system I/O	Browser file picker, drag-and-drop
Persistence	HDF5 and `.dlcapsule` files on disk	HDF5 and `.dlcapsule` download / upload; Desktop History is not preserved when DataLab-Web opens and saves an HDF5 workspace
Memory	Limited by system RAM (64-bit)	~2 GB WebAssembly heap by default; an optional on-disk storage mode lifts this limit entirely, bounding the working set by available disk space instead of RAM
Responsiveness	Computations run in separate worker processes; the interface stays responsive and processings can be cancelled	The Python runtime lives in a Web Worker, off the interface thread; the interface stays responsive and batch processings can be cancelled at object boundaries
Languages	English and French	English and French (browser language detected automatically)
Remote control	XML-RPC and FastAPI Web API	In-browser proxy and optional `postMessage` SDK

Both editions use the Sigima computation engine and can exchange HDF5 workspace files, but the features and information they preserve are not identical. DataLab-Web provides a curated subset of Sigima operations and does not preserve Desktop History when it opens and saves an HDF5 workspace. Check the edition-specific documentation before moving a workspace between them.

> **Note**
>
> Plugins are largely portable between the desktop application and DataLab-Web, with a few constraints related to the browser runtime. See Plugins and DataLab-Web for details.

1.5 Key features

This page presents briefly DataLab key features.

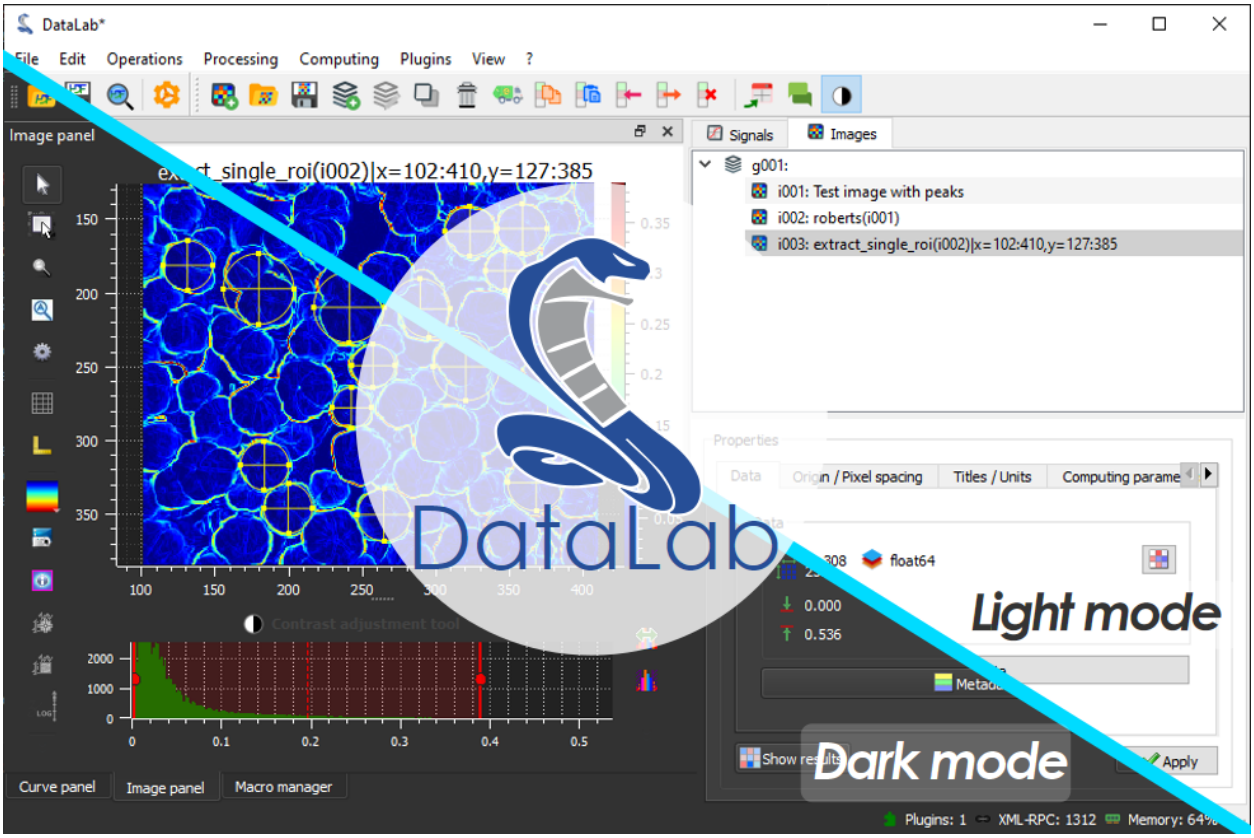


Fig. 5: DataLab supports dark and light mode. The color mode may be chosen in the "General" tab of the Settings: by default ("auto" mode), it follows your platform settings (this is handled by the guidata package, and may be overridden by setting the QT_COLOR_MODE environment variable to dark or light).

1.5.1 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
✓	✓	Persistence of settings in workspace
	✓	Distribute images on a grid
✓	✓	Single or superimposed views

1.5.2 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
✓	✓	Logarithm, exponential…
✓		Square root, power
✓	✓	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)
	✓	Horizontal and vertical projections
	✓	Pixel binning
✓		Linear calibration
	✓	Polynomial calibration
✓	✓	Normalization, Clipping, Offset correction
✓		Reverse X-axis
✓		Replace X by other signal's Y (calibration)
✓		X-Y mode
	✓	Thresholding (manual, Otsu…)
✓	✓	Gaussian filter, Wiener filter
✓	✓	Moving average, moving median
✓	✓	FFT, inverse FFT, Power/Phase/Magnitude spectrum, Power Spectral Density
✓		Interpolation
✓	✓	Resampling
✓		Detrending
✓		Interactive fit: Gauss, Lorentz, Voigt, polynomial, CDF…
✓		Interactive multigaussian fit
✓		Frequency filters (low-pass, high-pass, band-pass, band-stop)
✓		Windowing (Hann, Hamming…)
✓		Stability analysis (Allan variance, Hadamard variance…)
	✓	Butterworth filter
	✓	Exposure correction (gamma, log…)
	✓	Restoration (Total Variation, Bilateral…)

continues on next page

Table 1 – continued from previous page

Signal	Image	Feature
	✓	Morphology (erosion, dilation…)
	✓	Edge detection (Roberts, Sobel…)
✓	✓	Analysis on custom ROI
✓		FWHM, FW @ $1/e^2$
✓		Pulse features extraction
✓		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…)

1.5.3 Workspace and traceability

- HDF5 workspaces, and an HDF5 browser to import data sets from any HDF5 or NeXus file (see HDF5 Browser)
- History panel to record, replay (silently or step by step) and duplicate processing chains (see History Panel)
- Provenance of results: how each result was obtained, checked against the data and verified by recomputation (see Provenance of results)
- Workspace capsules (`.dlcapsule`) to exchange a workspace with the provenance of its processing (see Workspace capsules)
- Welcome page, Applications panel and command palette (see Overview)
- Optional AI assistant (see AI Assistant)

1.5.4 Extensibility

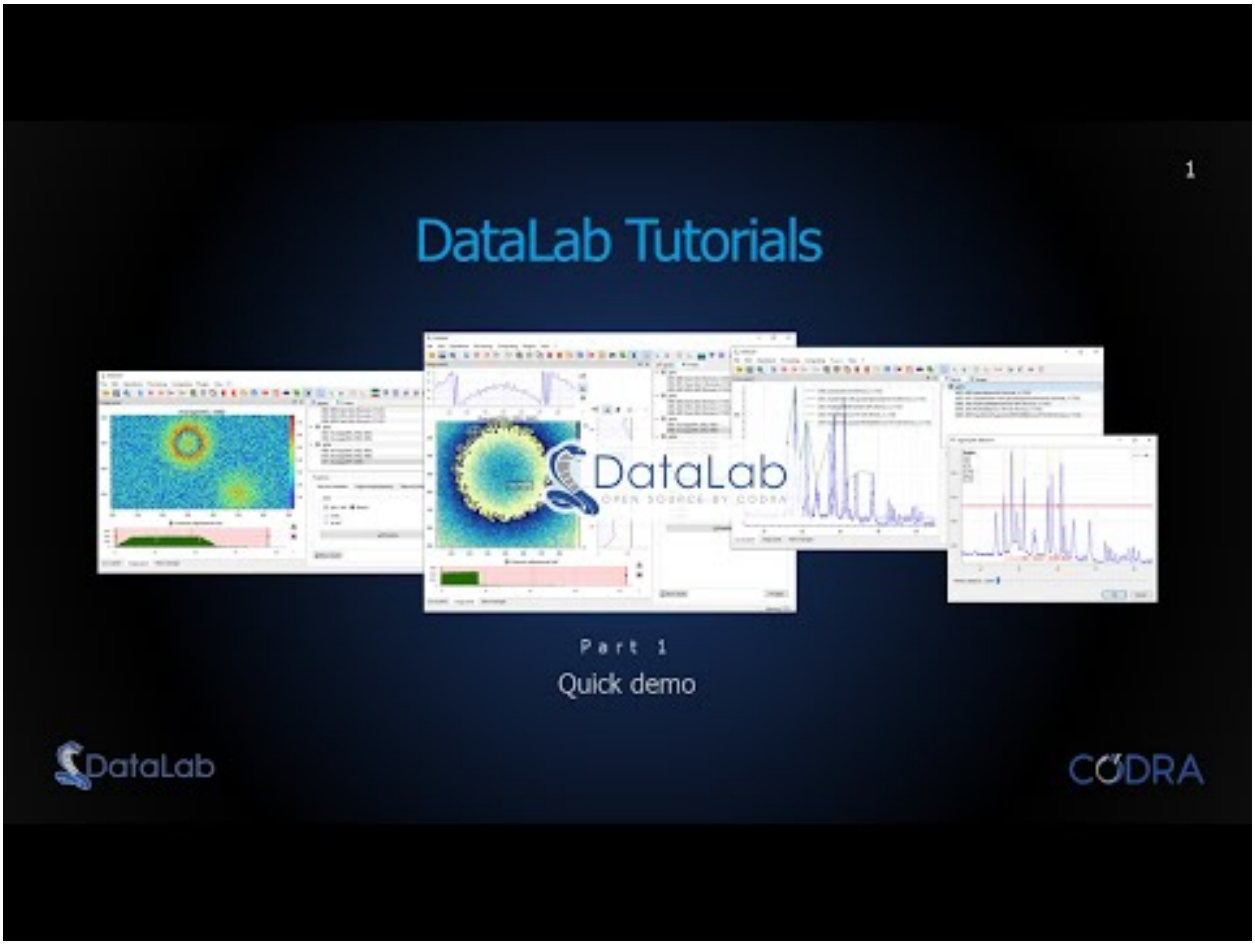
- Plugins adding processing features, file formats or complete applications, reloaded without restarting DataLab (see Plugins)
- DataLab plugin catalog: official and community plugins, installed in one click from DataLab
- Plugin generator: a working plugin in a single Python file, ready to publish in the catalog (see From an idea to the plugin catalog)
- Macro-commands, remote control and Web API to automate DataLab from Python, Jupyter notebooks or any application

1.6 Tutorials

1.6.1 Video Tutorials

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

Quick demo



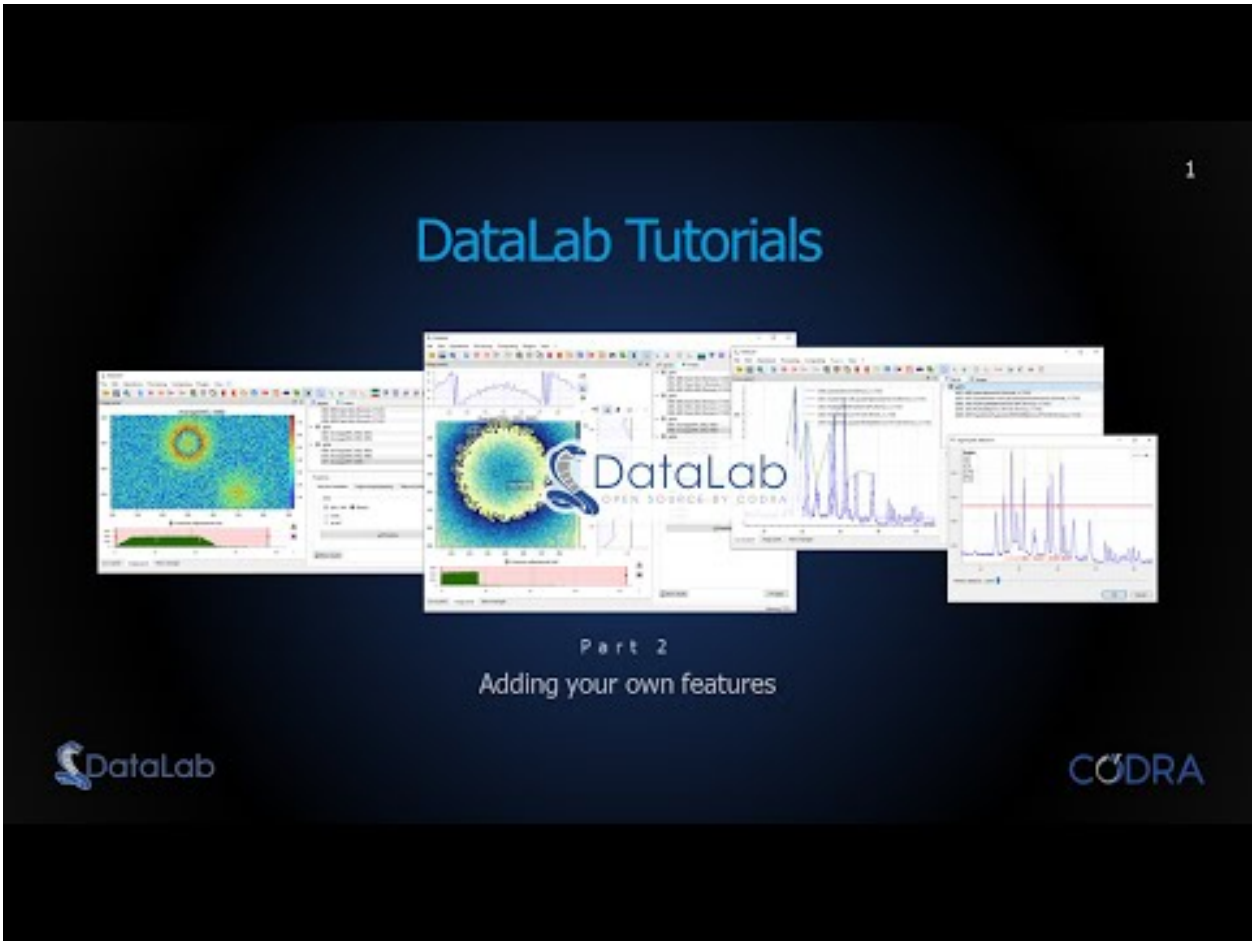
> **Warning**
>
> This video shows an older version of DataLab. Some features may have changed since then. Please refer to the documentation for the most up-to-date information.

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.

1.6.2 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.

Spectrum analysis: denoising, baseline correction and peak fitting

This example shows how to analyze 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 signal…", 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.

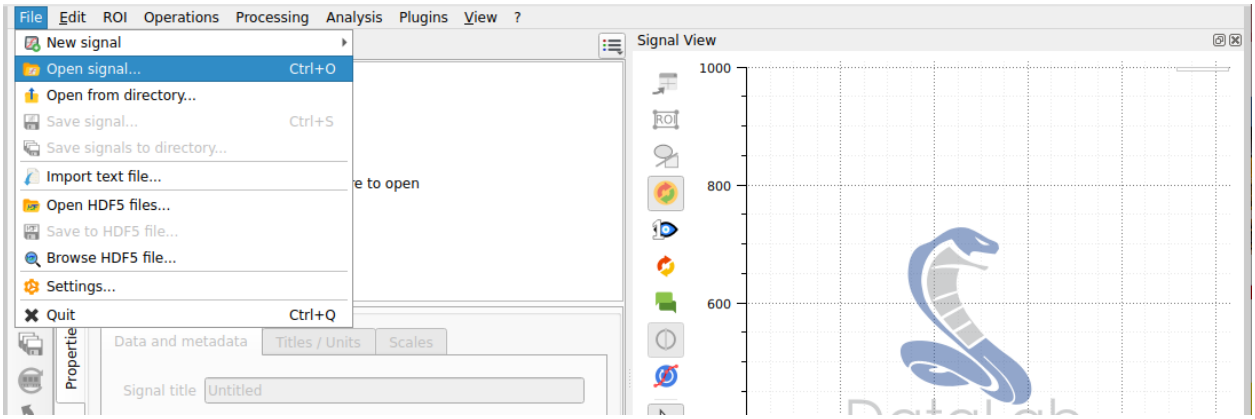


Fig. 6: The "File > Open signal…" menu to open a spectrum file.

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 diffraction angle 2θ (in degrees), 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 ("Regions of Interest") will appear—click "OK" to extract the region of interest. A new signal containing the ROI will be created and displayed in the main window.

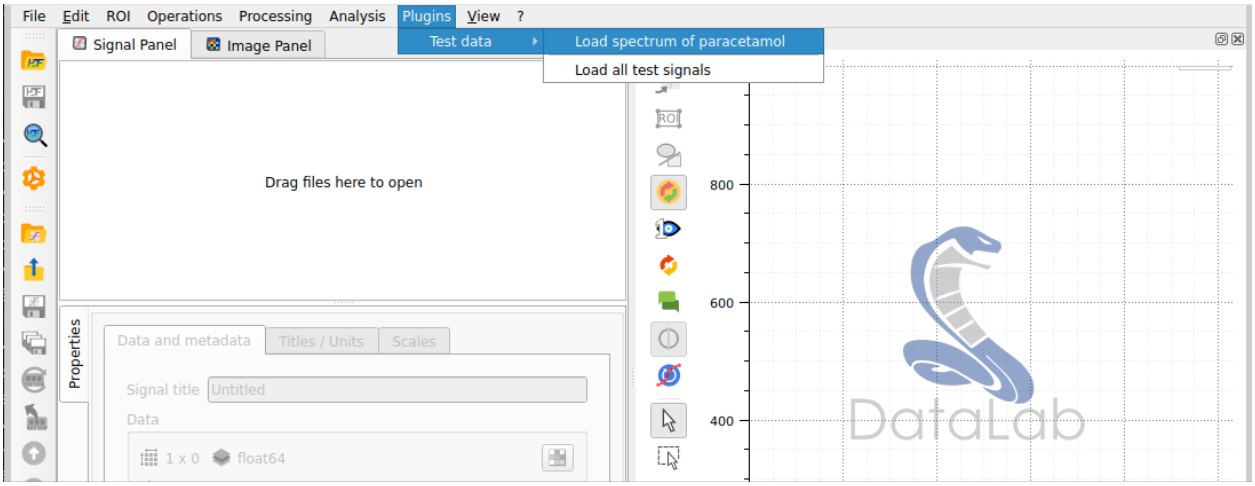


Fig. 7: The “Plugins > Test data > Load spectrum of paracetamol” plugin to generate the test spectrum for this tutorial.

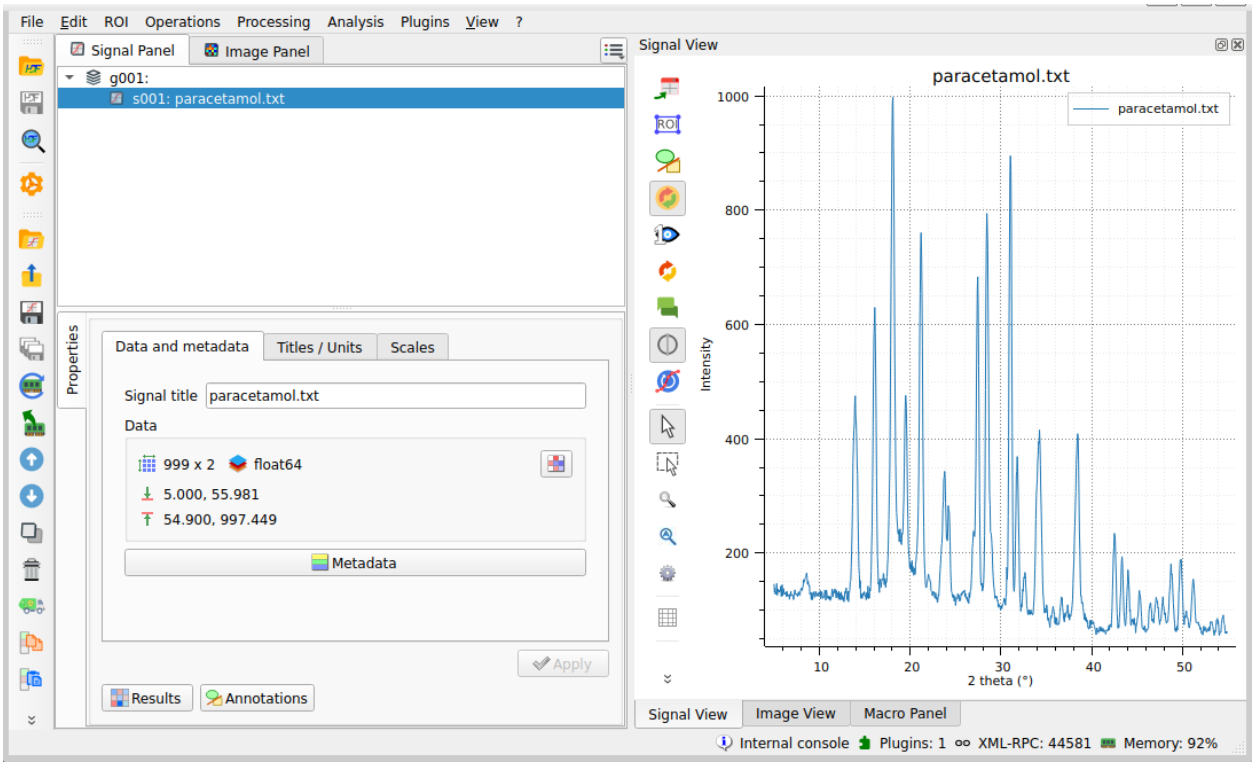


Fig. 8: The spectrum displayed on the “Signal View” panel.

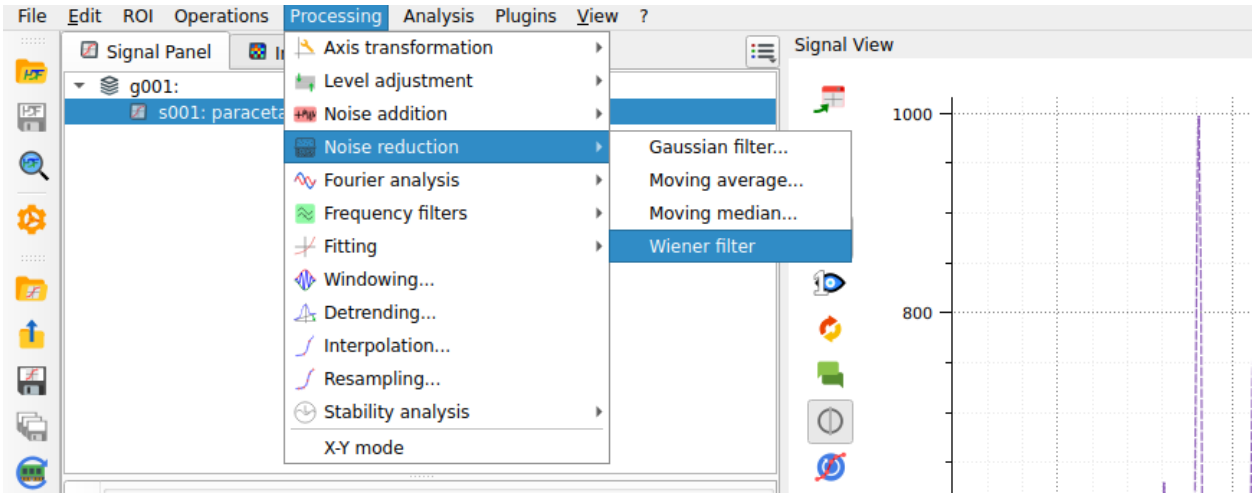


Fig. 9: The “Processing > Noise reduction > Wiener filter” menu option.

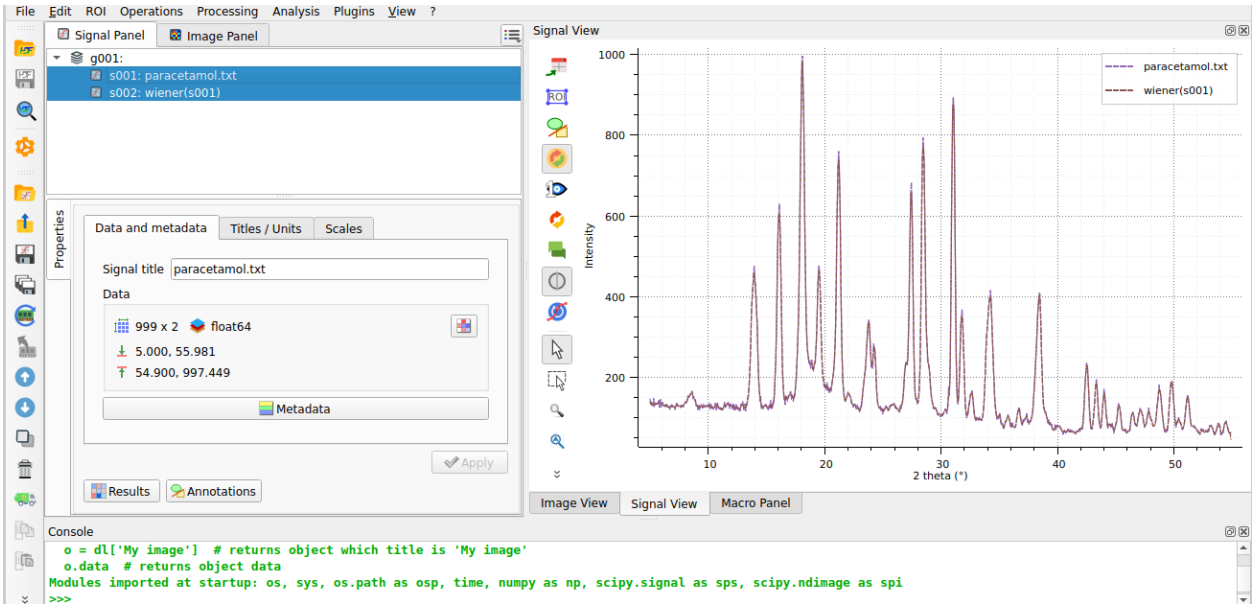


Fig. 10: The result displayed in the main window.

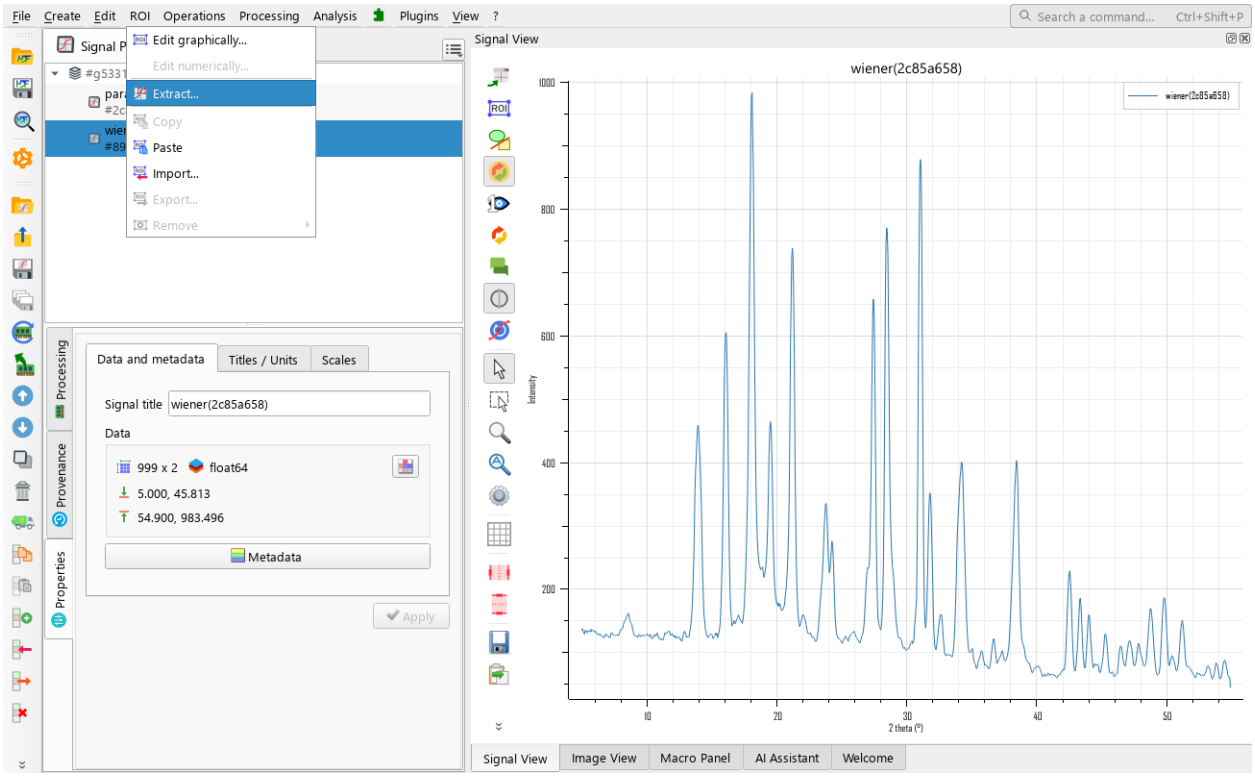


Fig. 11: The “ROI > Extract…” menu.

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.

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.

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:

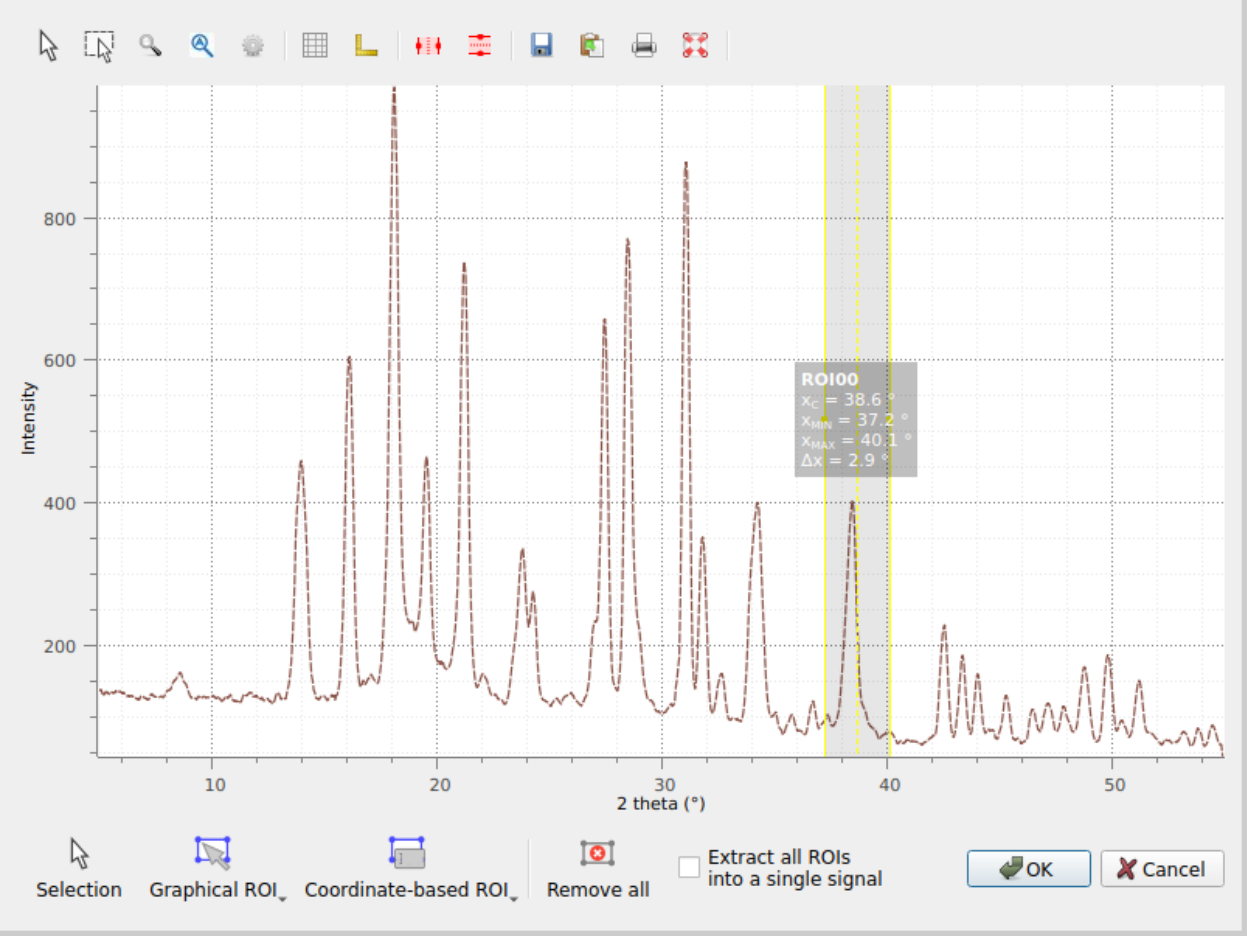


Fig. 12: The “Regions of interest” dialog box displayed.

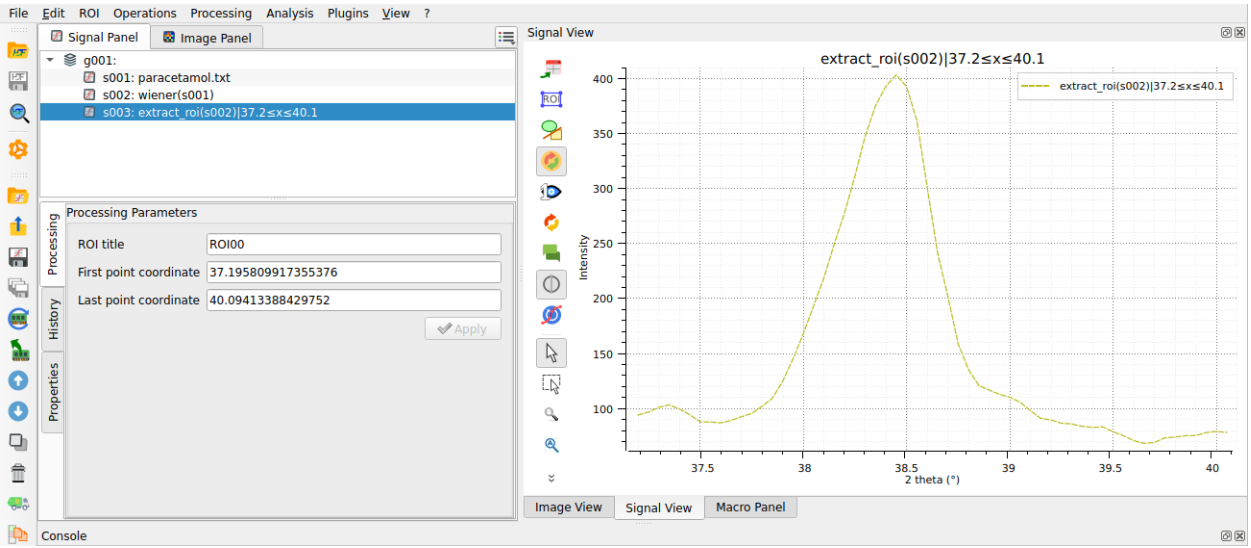


Fig. 13: The region of interest displayed in the main window.

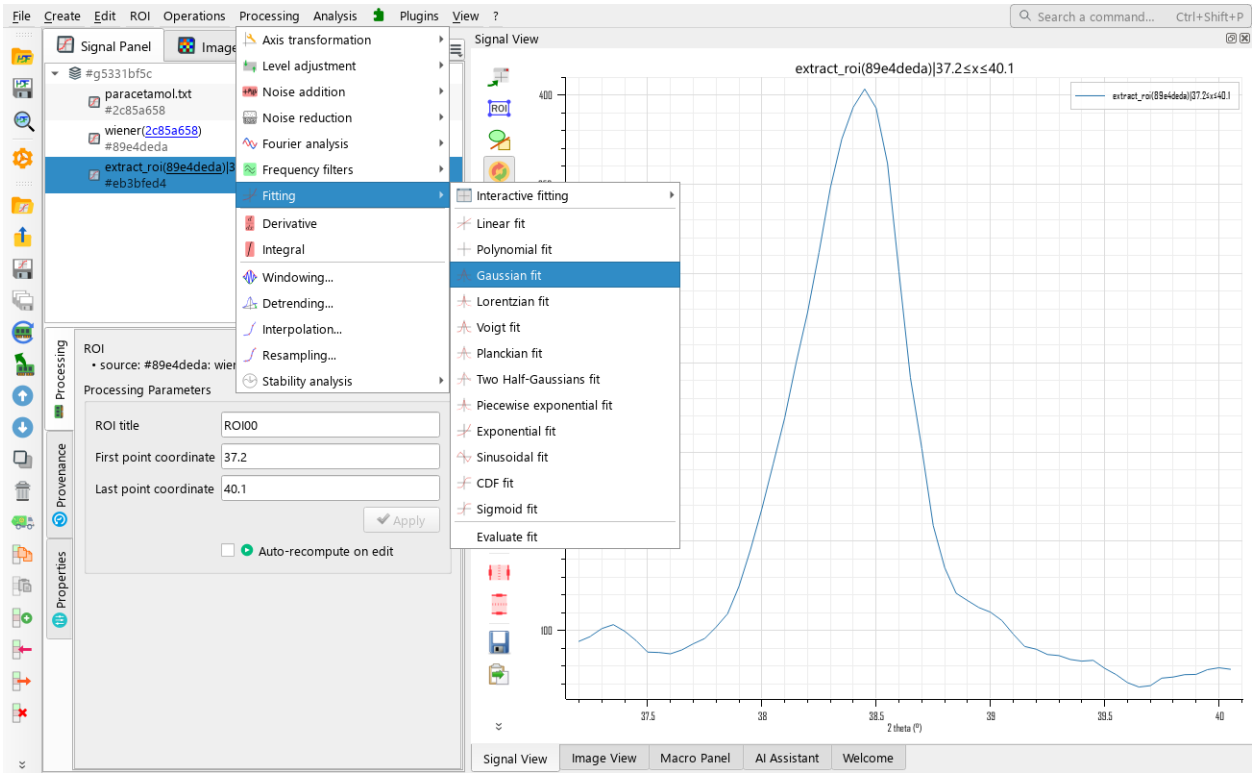


Fig. 14: Fit a Gaussian model to the extracted peak with “Processing > Fitting > Gaussian fit” (automatic fit, without dialog box; an interactive version is available in the “Processing > Fitting > Interactive fitting” submenu).

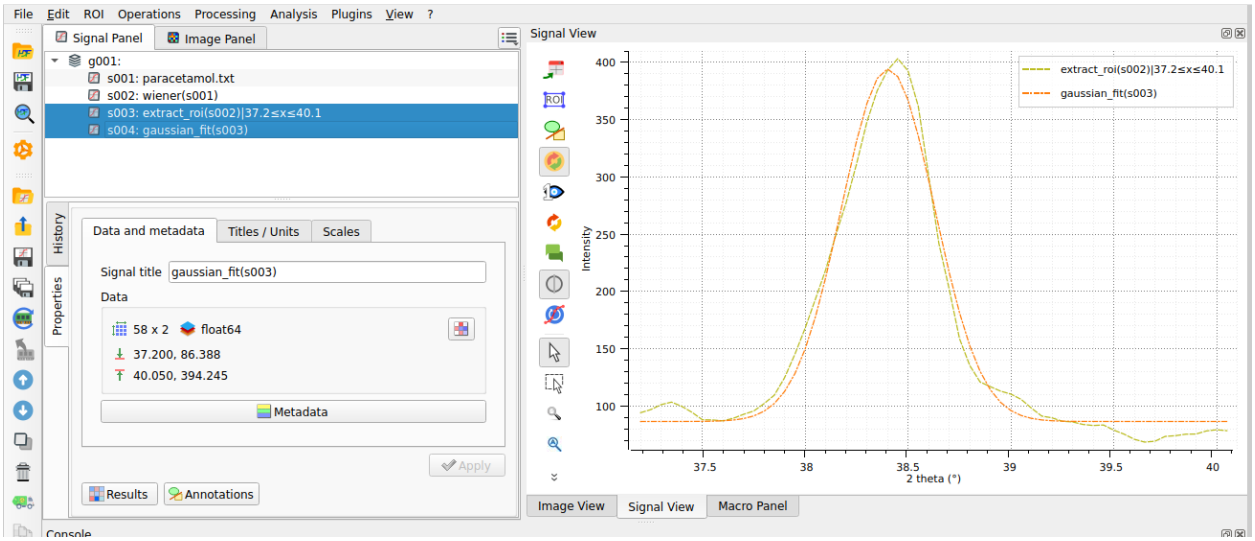


Fig. 15: The result of the fit displayed in the main window.

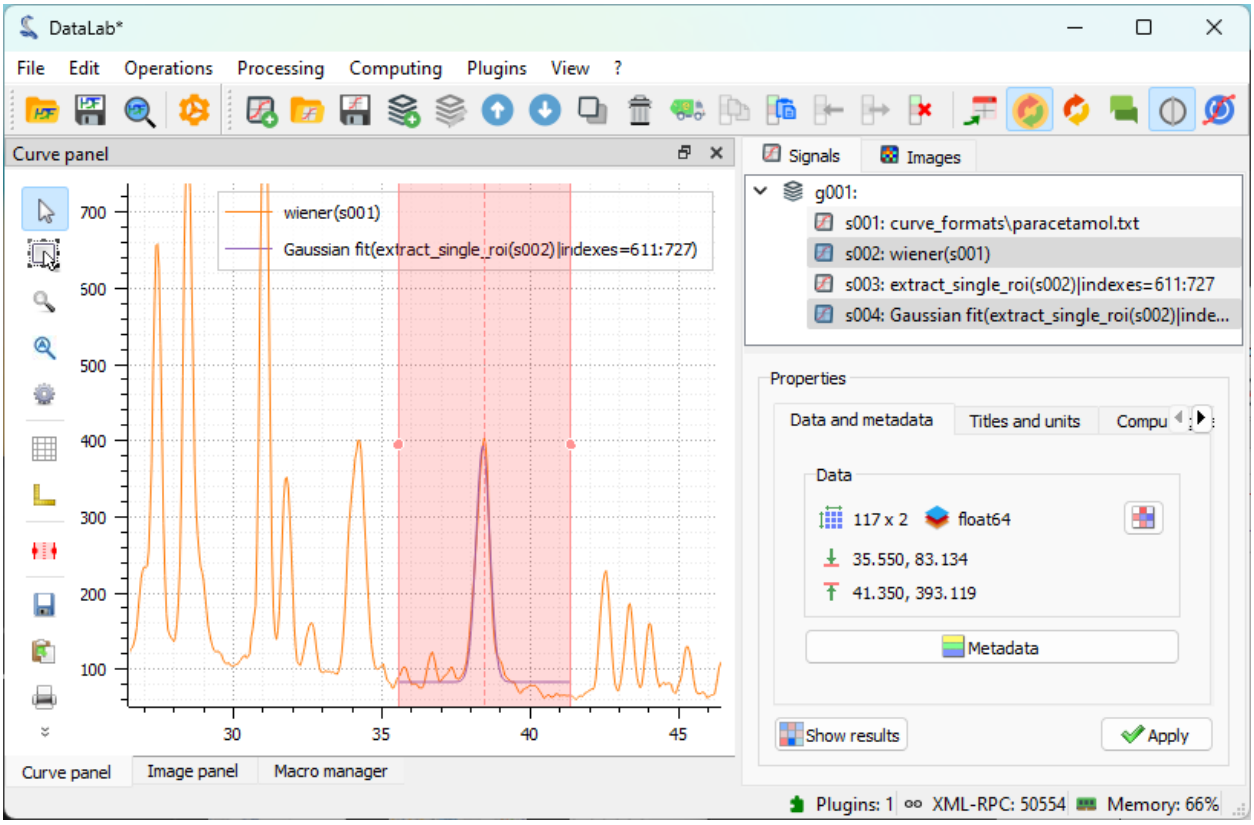


Fig. 16: Both the full spectrum and the fit are selected in the "Signals" panel, so that both are displayed in the visualization panel on the left. This allows for easy visual comparison if needed for the analysis.

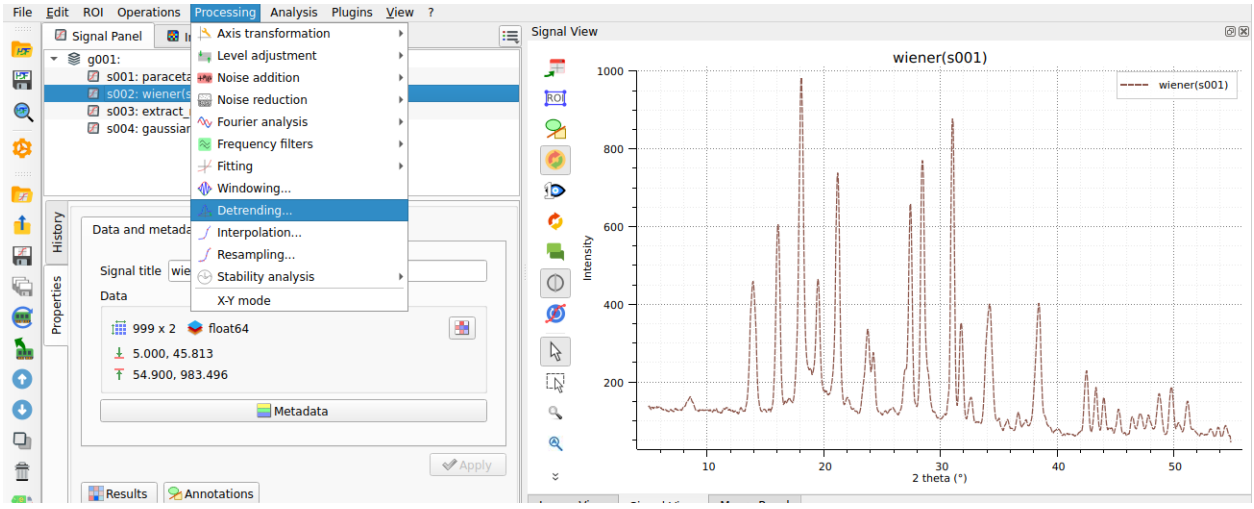


Fig. 17: The "Processing > Detrending" feature.

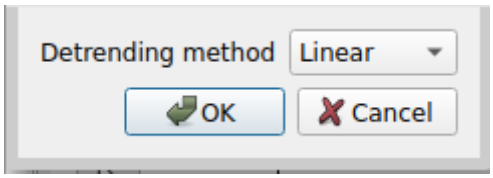


Fig. 18: We choose a linear detrending method, and we click on “OK”.

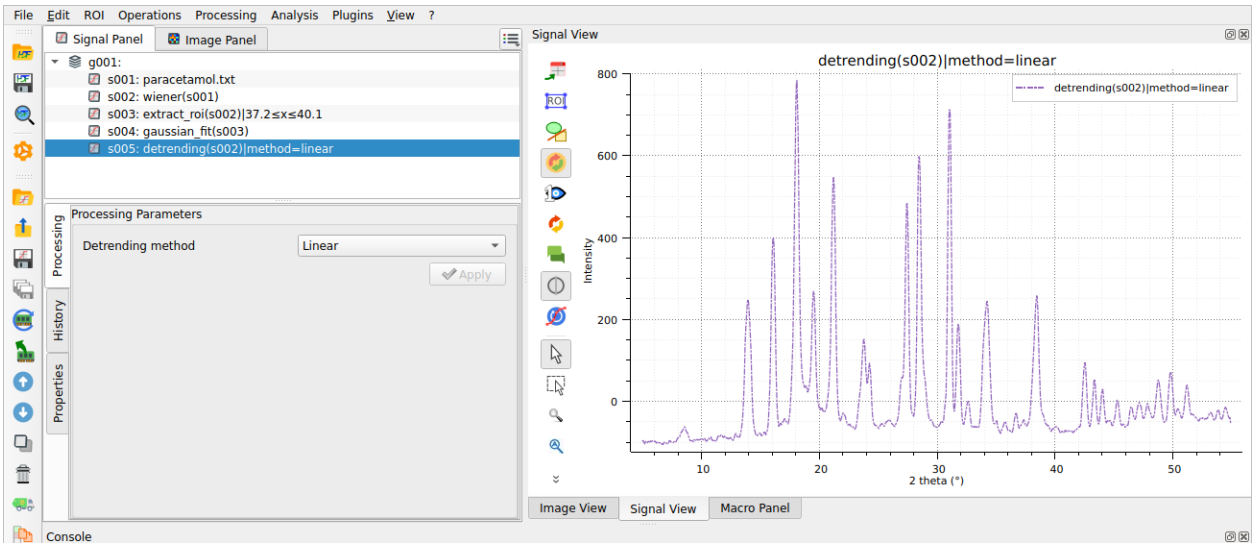


Fig. 19: The result of the detrending displayed in the main window.

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 > Interactive 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 extract peak positions as a result table attached to the signal.

After adjusting the peak detection parameters (using the same dialog as for the multi-Gaussian fit), click “OK”. The detected peak positions are stored as an XY-markers result and overlayed on the spectrum.

To rebuild the historical sticks signal — a delta function at each detected peak position — apply Operations Create signal from markers table… to the spectrum: this creates a child signal whose samples are the `(x, y)` couples of the markers table, rendered as sticks.

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 files…” (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.

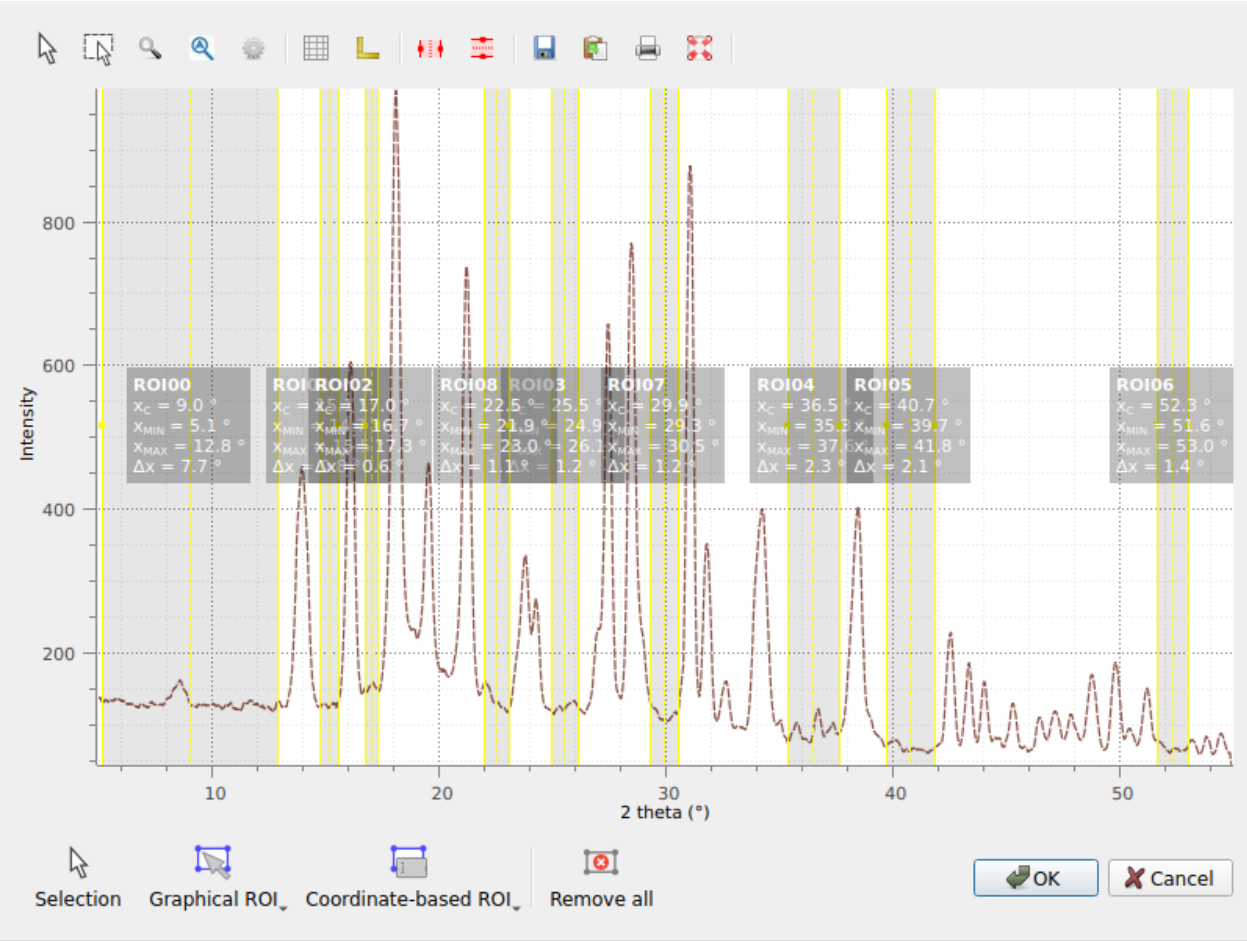


Fig. 20: We select the regions corresponding to the regions without peaks.

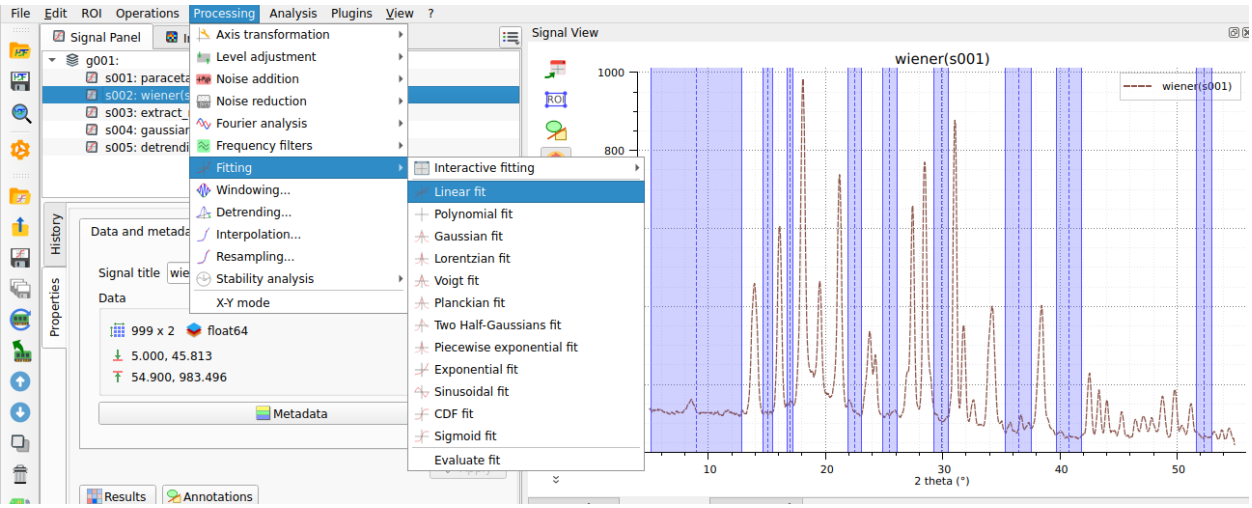


Fig. 21: A linear fit is performed only on the selected regions.

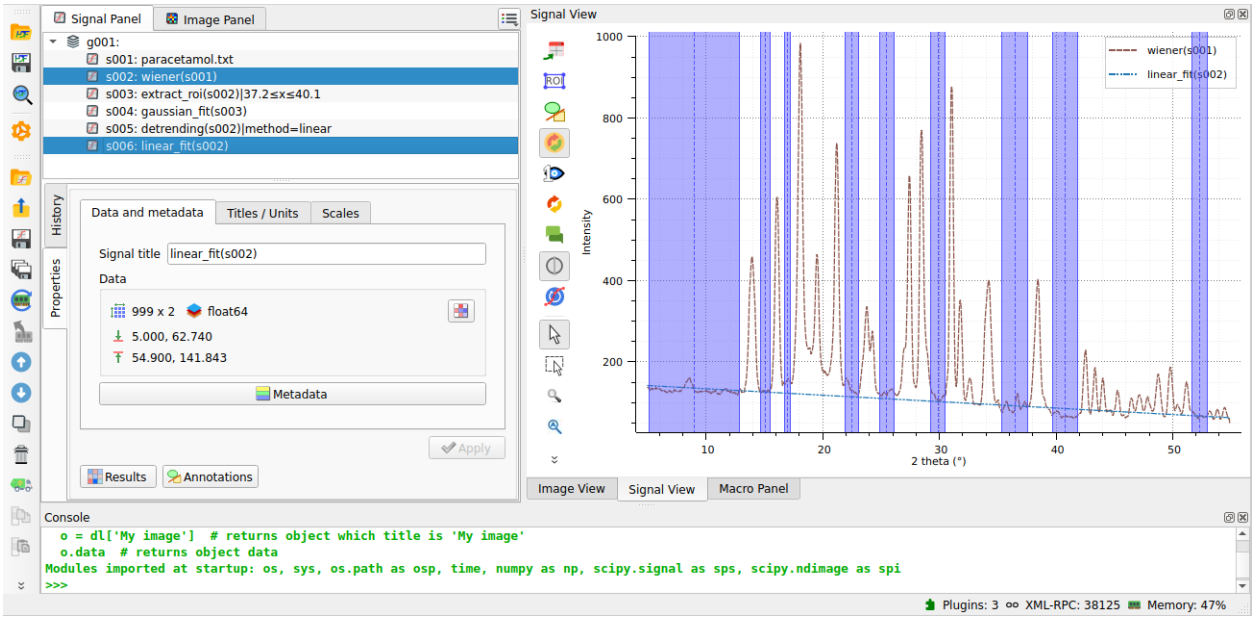


Fig. 22: The linear baseline obtained from the fit is shown.

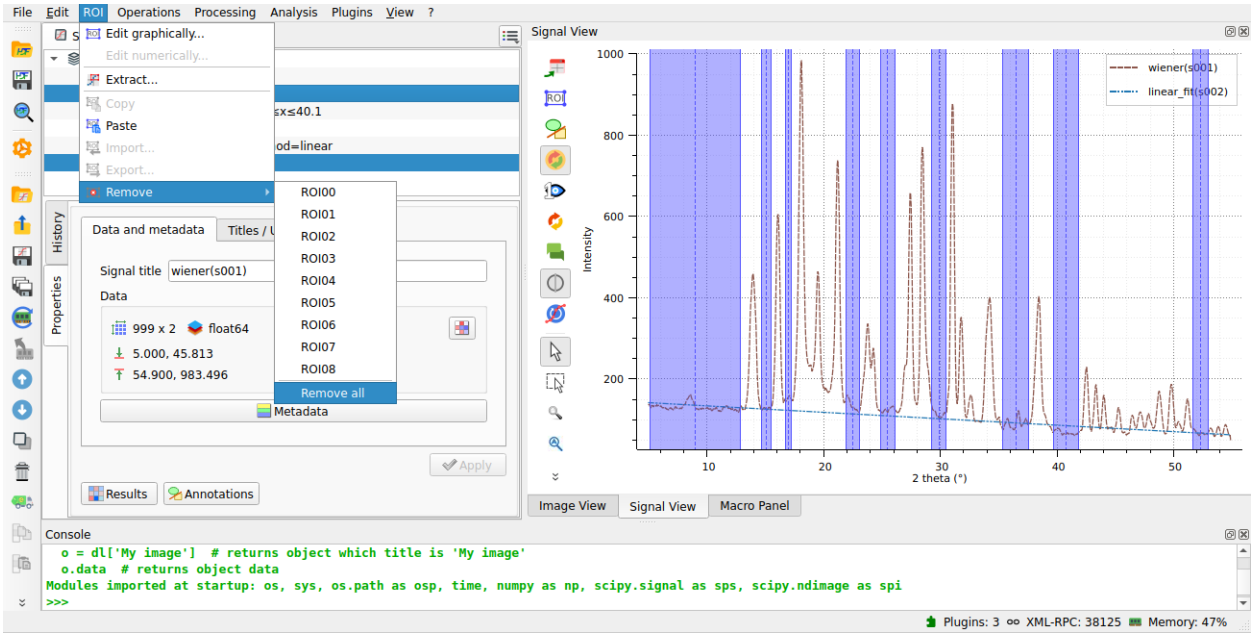


Fig. 23: We delete the rois to apply the detrending on the whole signal.

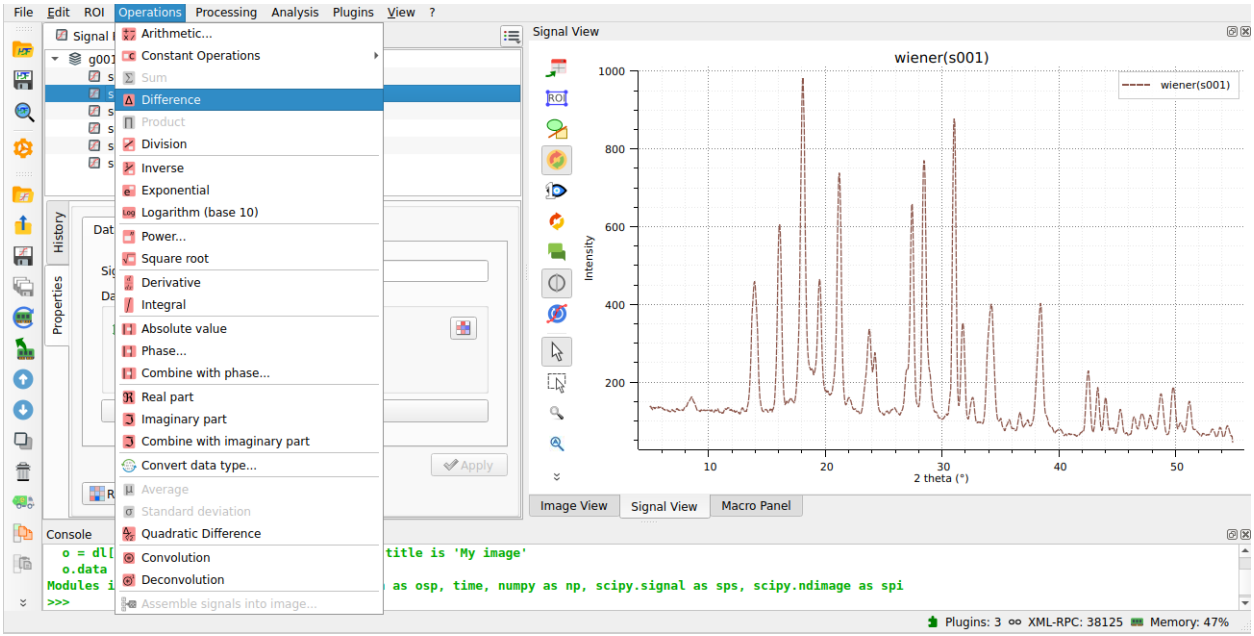


Fig. 24: We use the "difference" operation to subtract the baseline from the original signal.

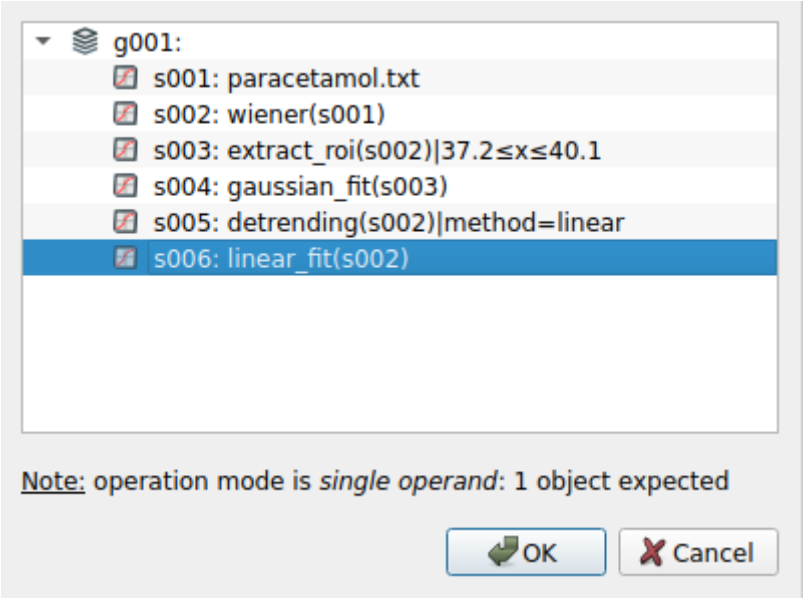


Fig. 25: We select the linear baseline to perform the subtraction.

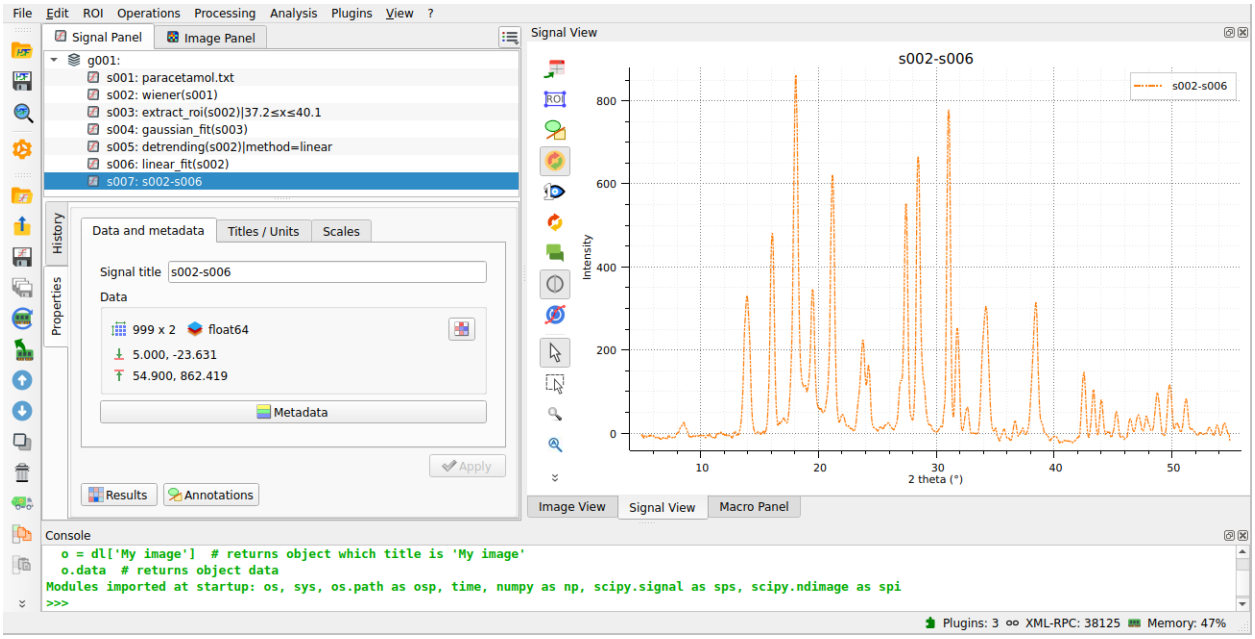


Fig. 26: The resulting detrended signal is shown, now with a correct baseline.

Automated detection of defects and particles on images

This example shows how to automatically detect blob-like features – defects, particles, spots – 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. Plugins are loaded from plugin directories, which are listed in the "Advanced" tab of the plugin configuration dialog ("Plugins > Configure plugins…"). We can add a new plugin by copying/pasting the plugin file in one of these directories.

> **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 reload the plugins with "Plugins > Reload plugins" (or restart 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).

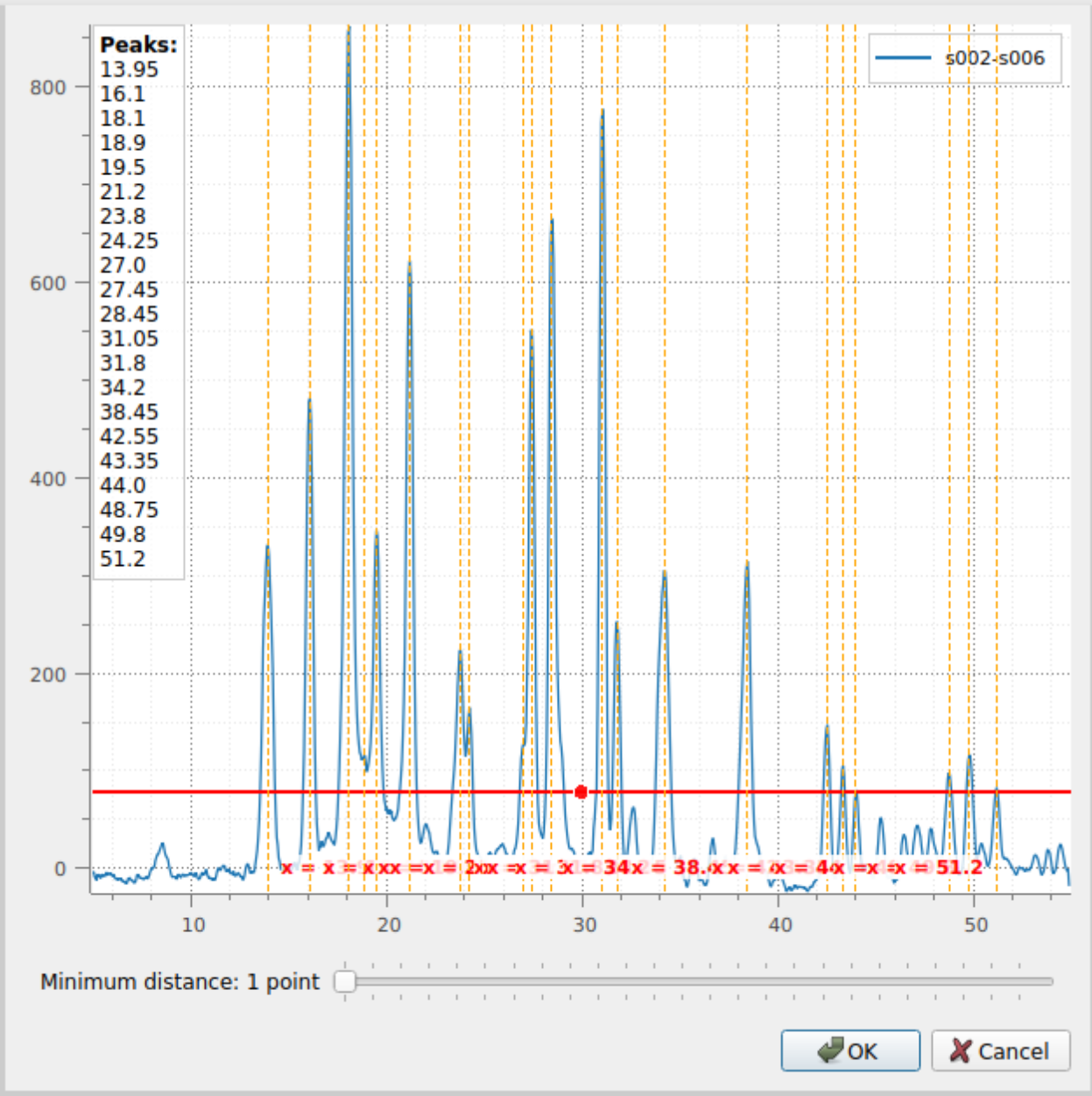


Fig. 27: First, the "Signal peak detection" dialog box appears. You can adjust the vertical cursor position to set the threshold for peak detection and specify the minimum distance between peaks. Then click "OK".

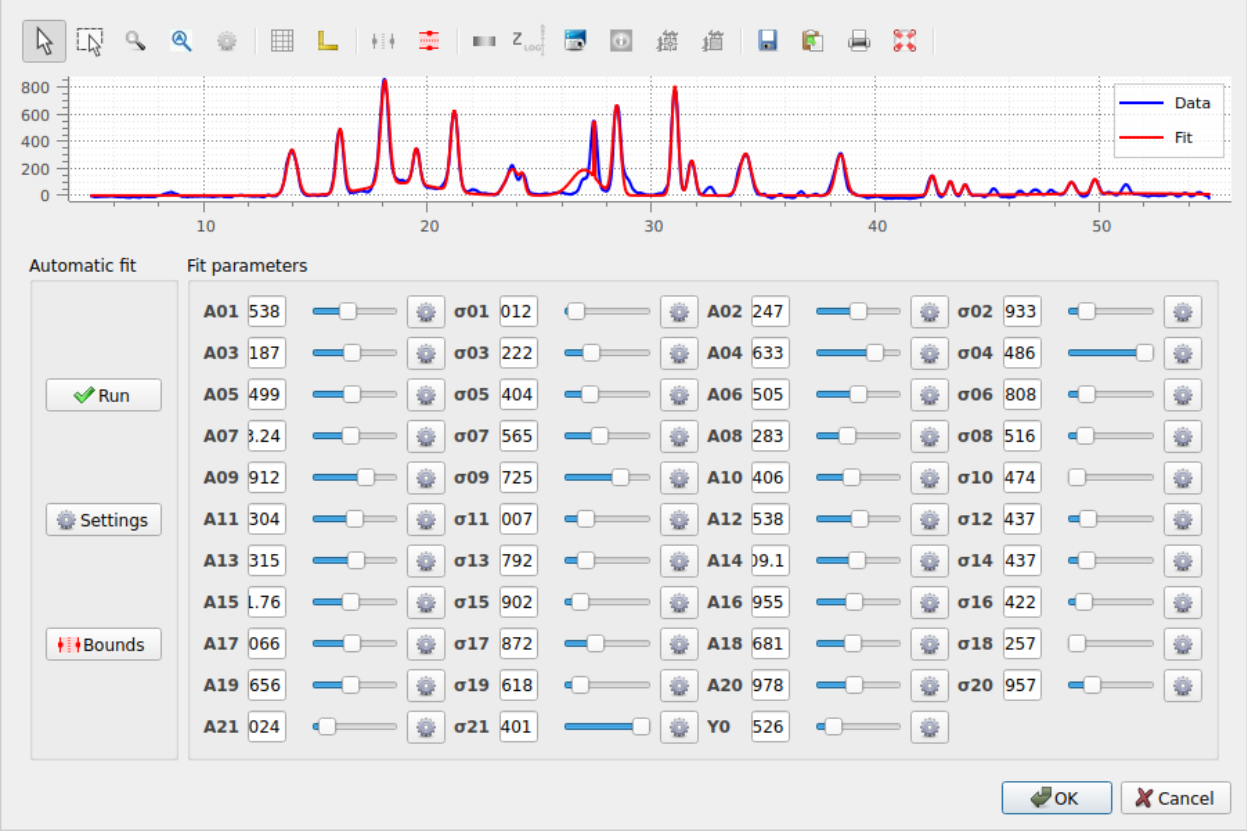


Fig. 28: The "Multi-Gaussian fit" dialog box appears. An automatic fit is performed by default. Click "OK" (or optionally adjust the parameters manually using the sliders, or modify the automatic fitting parameters).

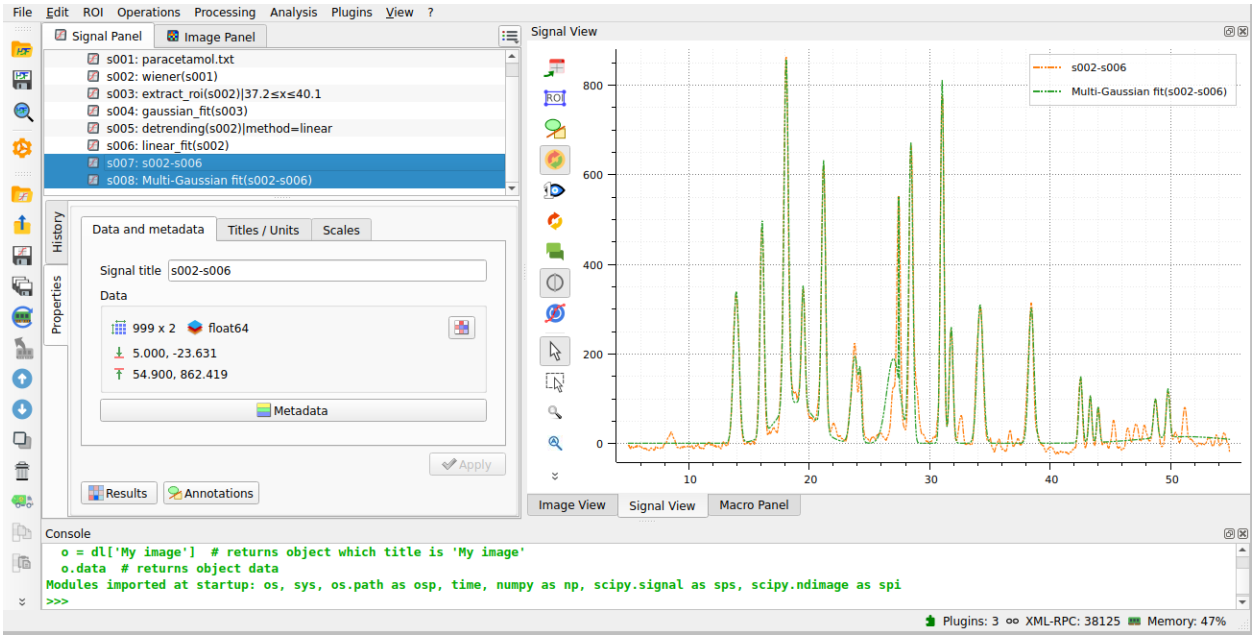


Fig. 29: The result of the fit is displayed in the main window. Here we selected both the spectrum and the fit in the "Signals" panel on the right, so both are displayed in the visualization panel on the left.

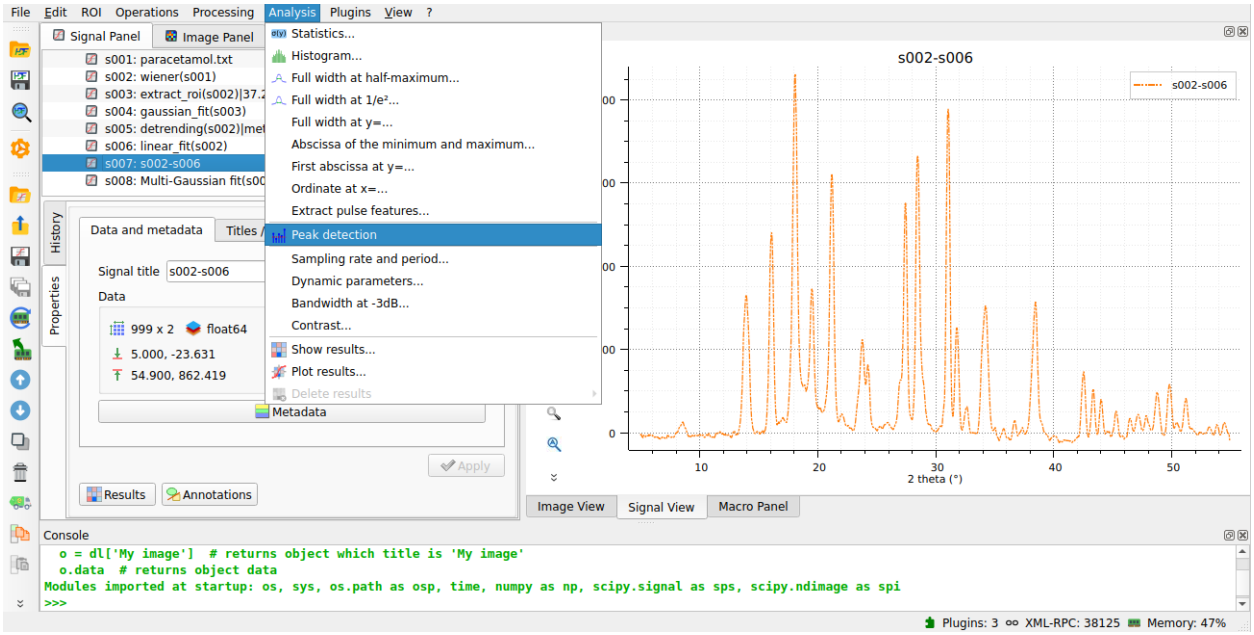


Fig. 30: Open the “Peak detection” window with “Analysis > Peak detection…”.

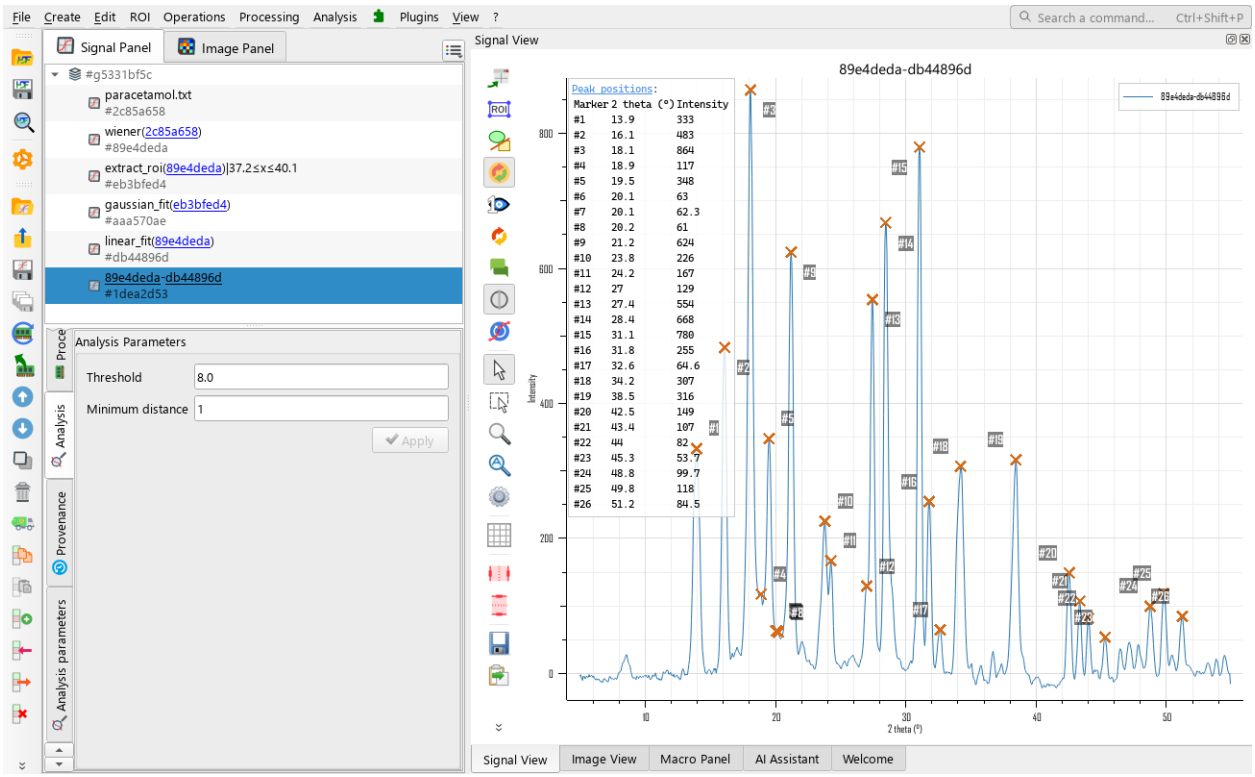


Fig. 31: The detected peak positions are shown as XY markers on the spectrum, with their coordinates in the result label. No additional signal is created.

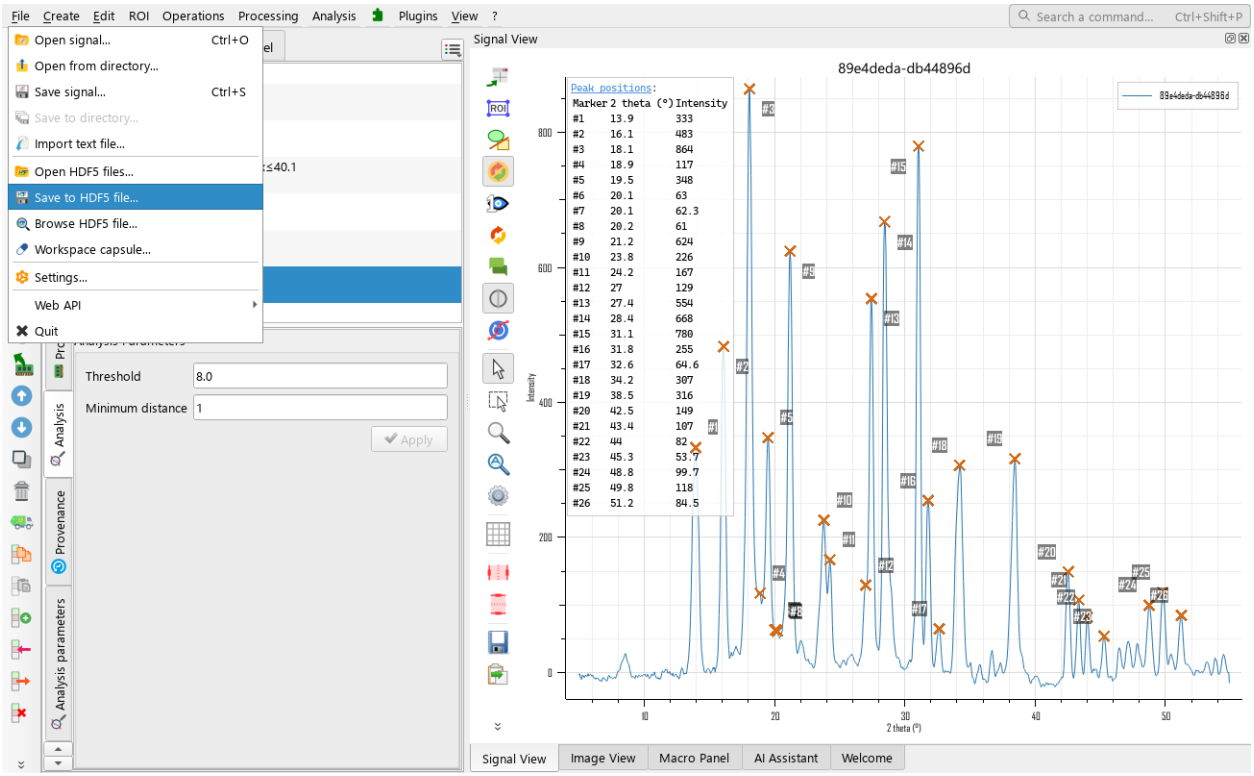


Fig. 32: Save the workspace to a file with “File > Save to HDF5 file…”, or the button in the toolbar.

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:

```
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
```

(continues on next page)

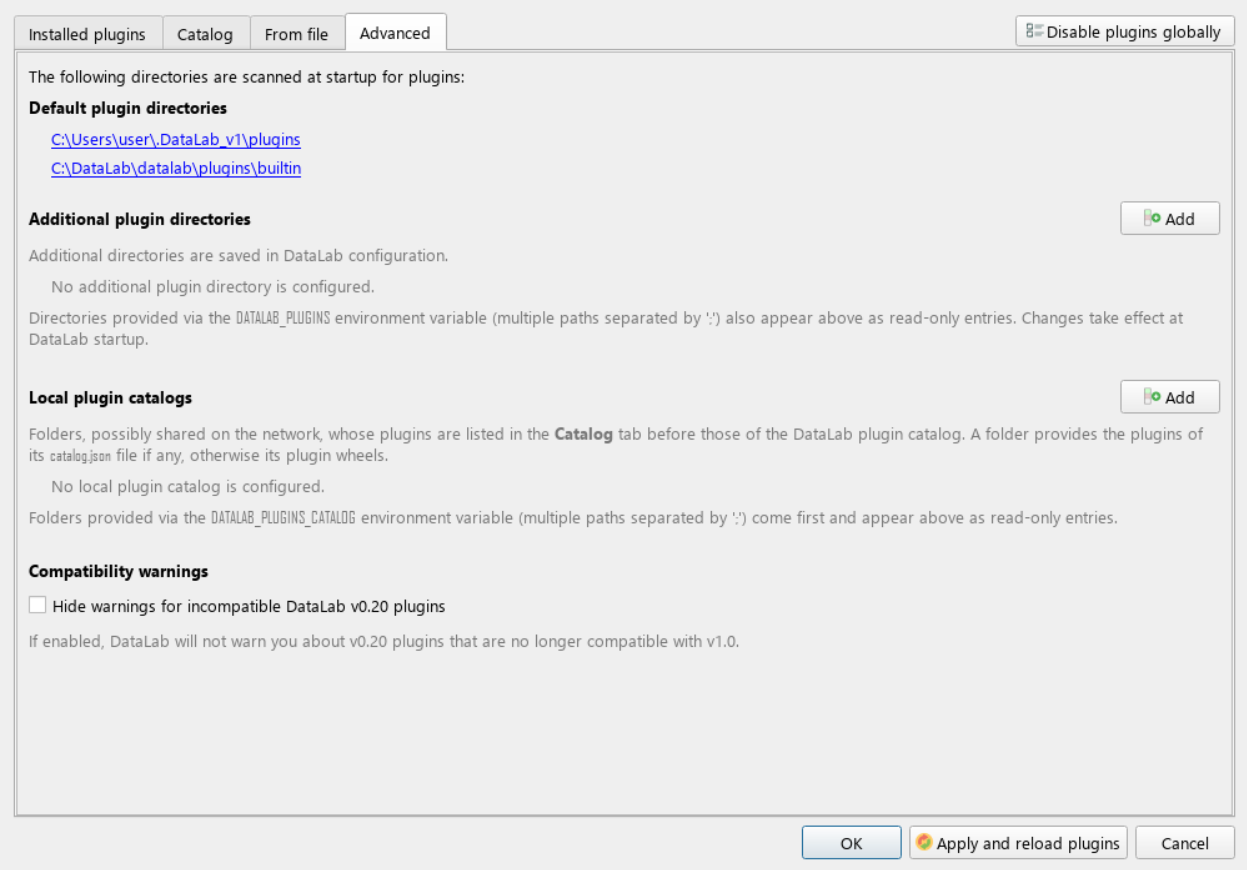


Fig. 33: The “Advanced” tab of “Plugins > Configure plugins…” lists the plugin directories (the paths shown here are examples).

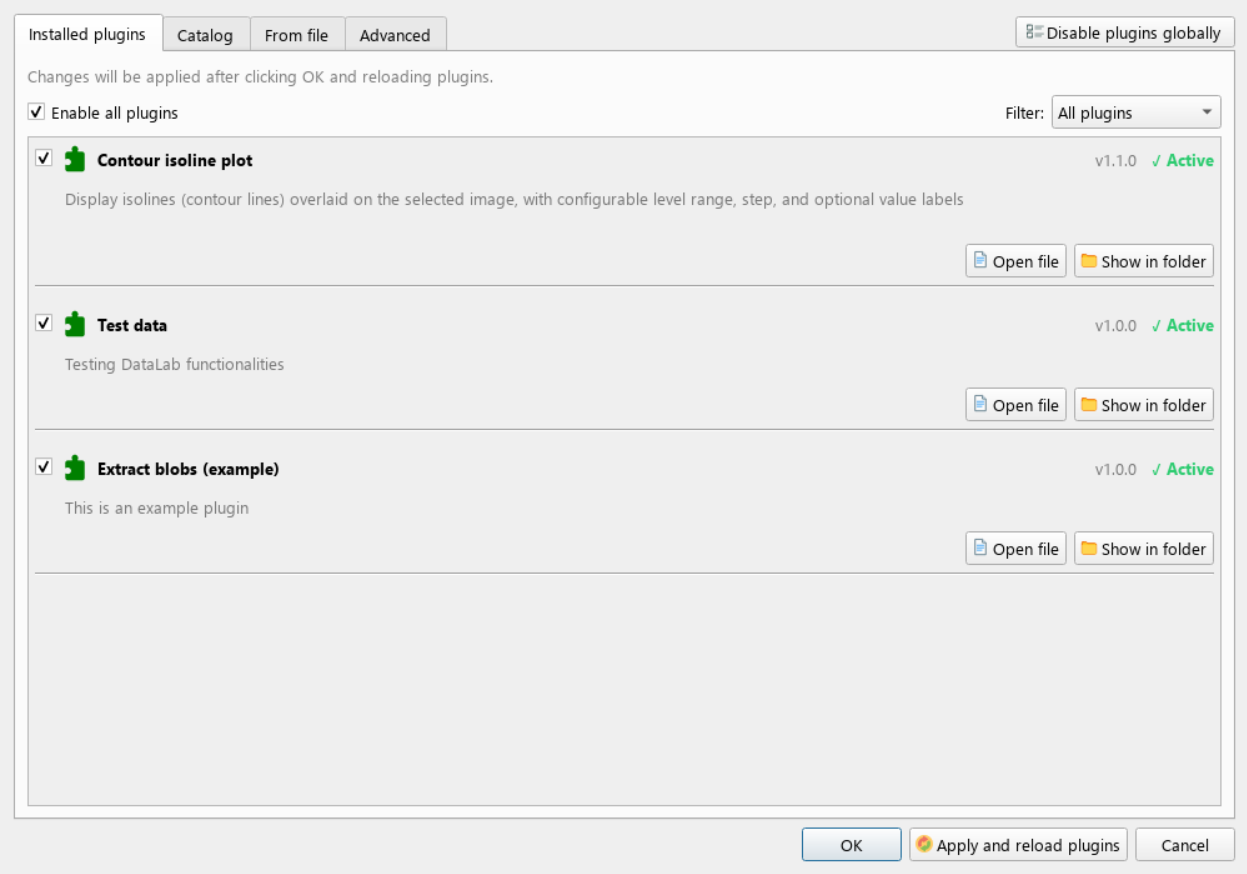


Fig. 34: The “Installed plugins” tab of “Plugins > Configure plugins…” shows the example plugin as active.

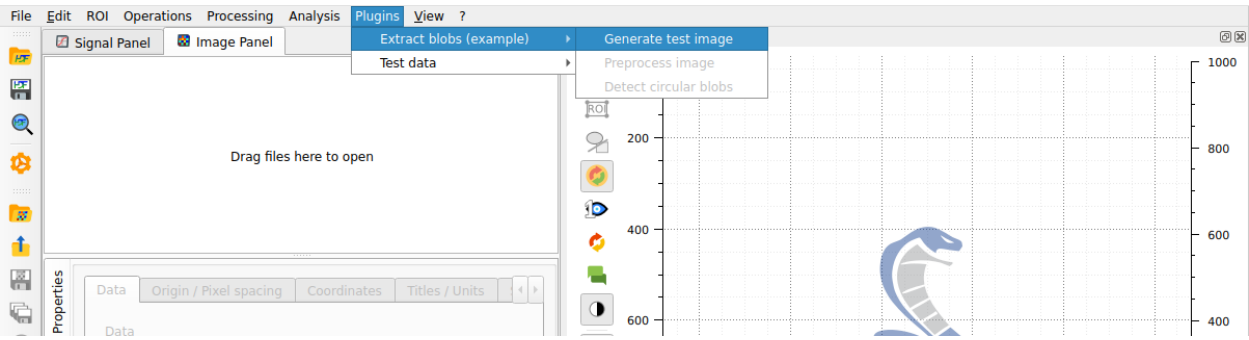


Fig. 35: Let’s click on “Extract blobs (example) > Generate test image”

(continued from previous page)

```
        obj = sigima.objects.create_image(
            newparam.title, data, units=("mm", "mm", "lsb")
        )
        self.proxy.add_object(obj)
```

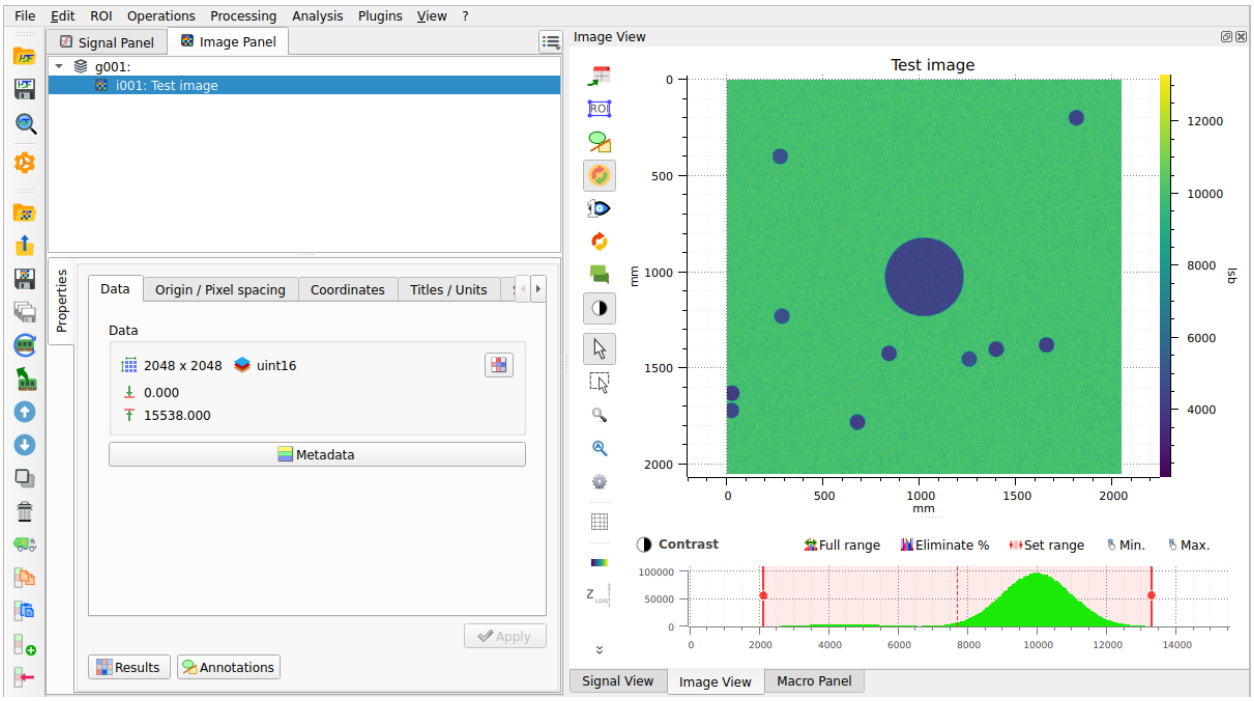


Fig. 36: The plugin has generated a test image, and added it to the "Images" panel. The image shows a few blobs, with a central bigger disk, and a noisy background.

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 improve the signal-to-noise ratio 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 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.

> **Note**
>
> OpenCV is an optional dependency of DataLab: it is included in the stand-alone version of DataLab but, when DataLab is installed with `pip`, the "Blob detection (OpenCV)" feature requires the `opencv` extra (`pip install`

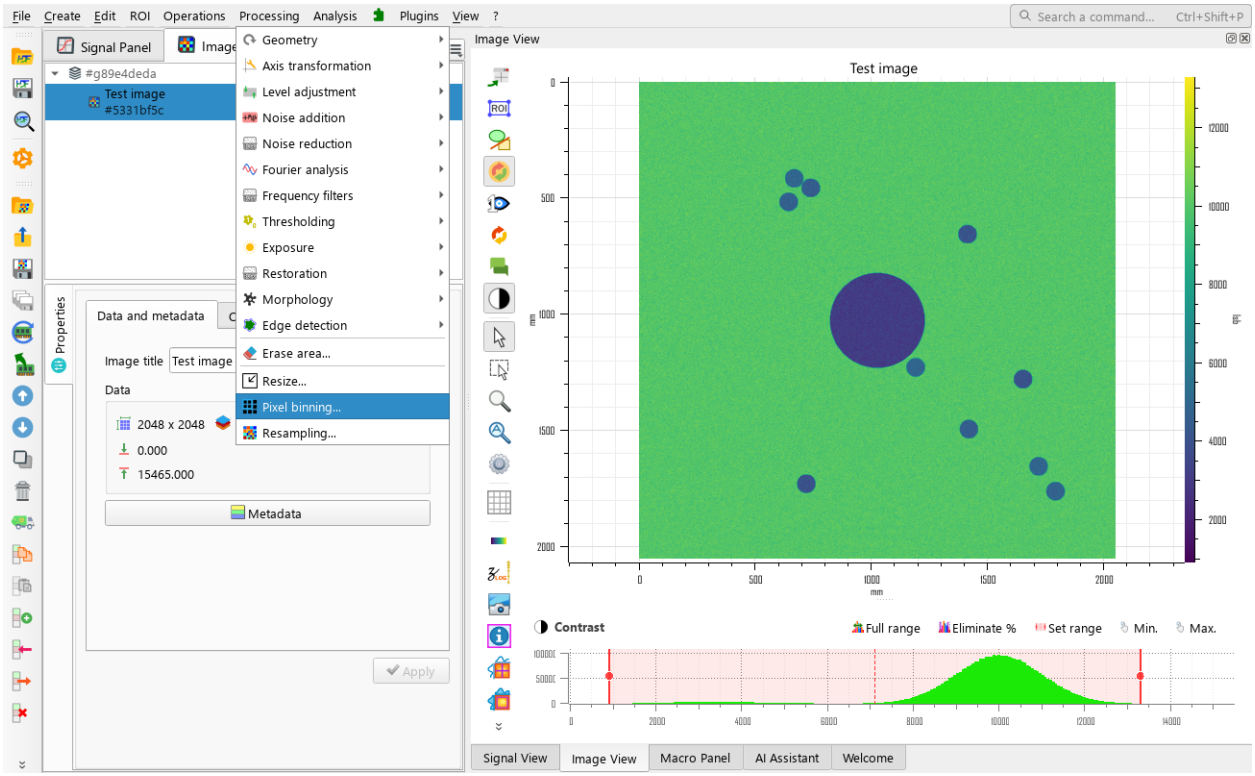


Fig. 37: Click on “Processing > Pixel binning…”.

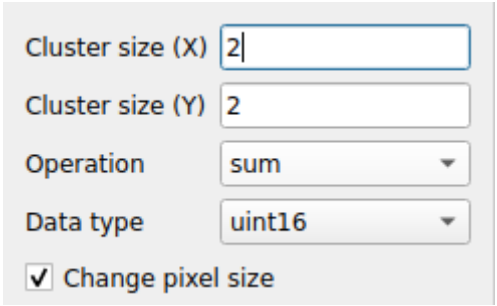


Fig. 38: The “Pixel binning” dialog opens, with several options. Set the cluster size to 2 for both axes, check the “Change pixel size” option and click on “OK”.

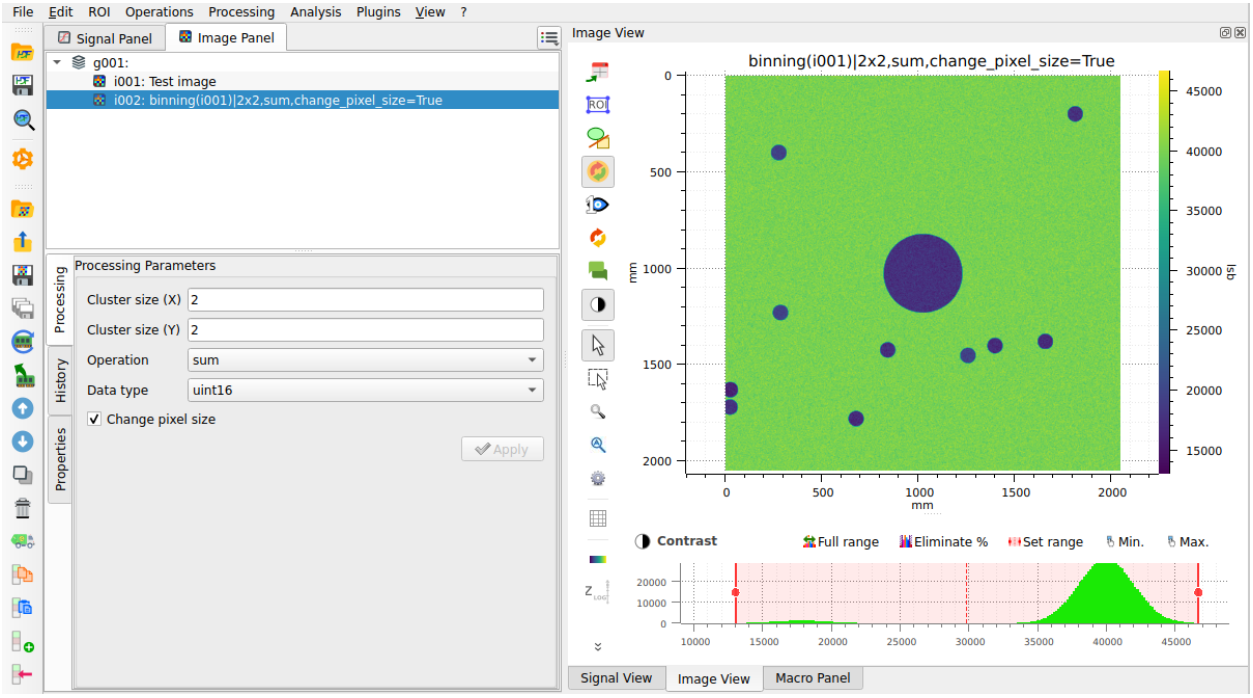


Fig. 39: The binned image is added to the "Images" panel. It is now easier to see the blobs (even if they were already quite visible on the original image: this is just an example), and the image will be faster to process.

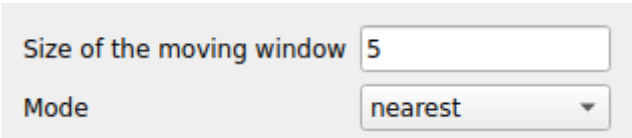


Fig. 40: Click on "Processing > Noise reduction > Moving median…" entry, and set the window size to `5`.

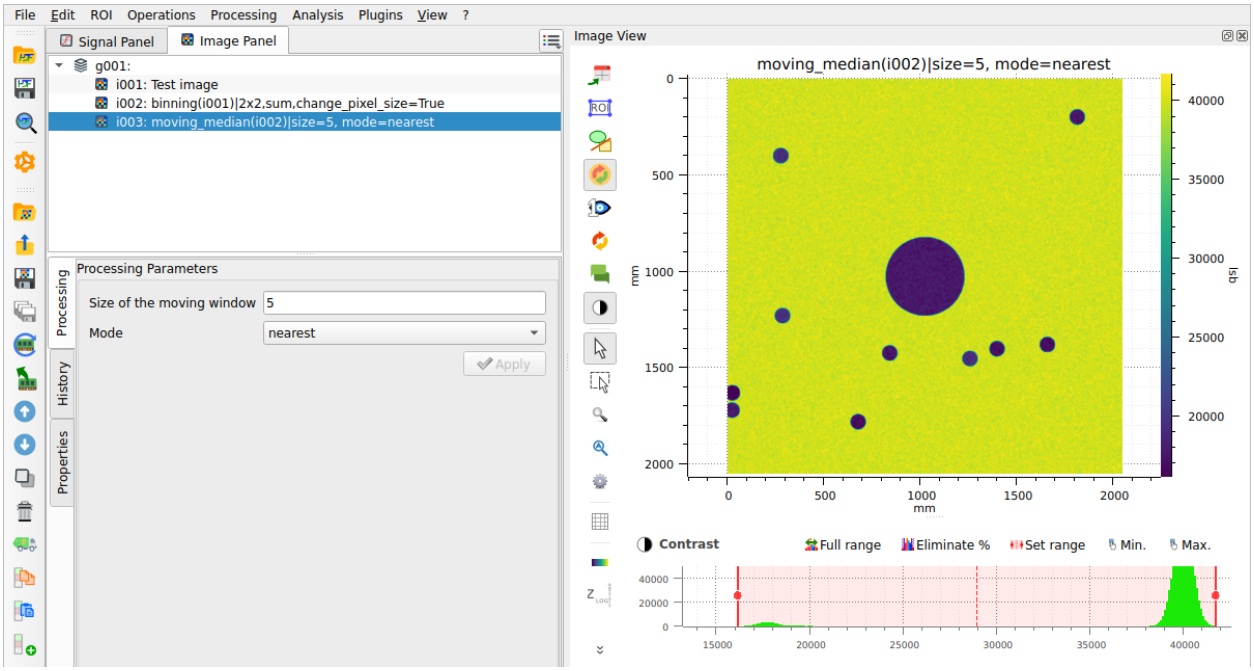


Fig. 41: The filtered image is added to the "Images" panel. Denoising is quite efficient.

`datalab-platform[opencv]`).

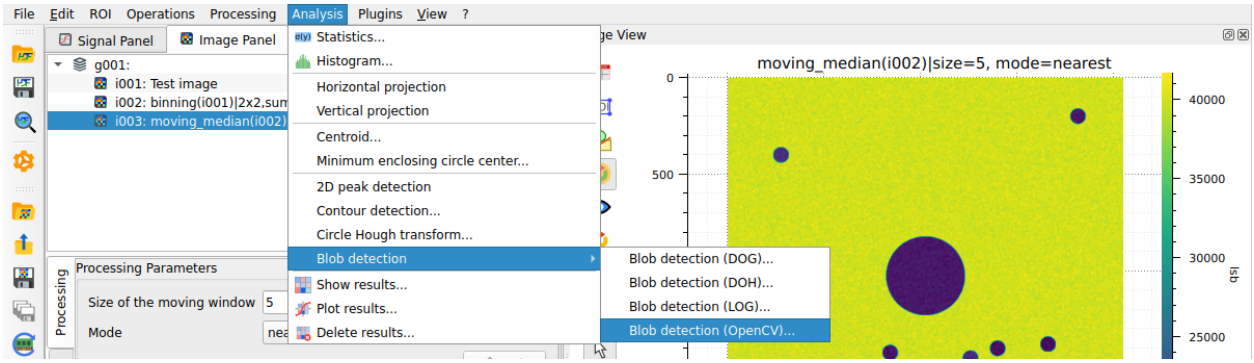


Fig. 42: Click on "Analysis > Blob detection > Blob detection (OpenCV)".

> **Note**
>
> If you want to show the analysis results again, you can select the "Show results…" entry in the "Analysis" menu.

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 files…" (or the button in the toolbar) to

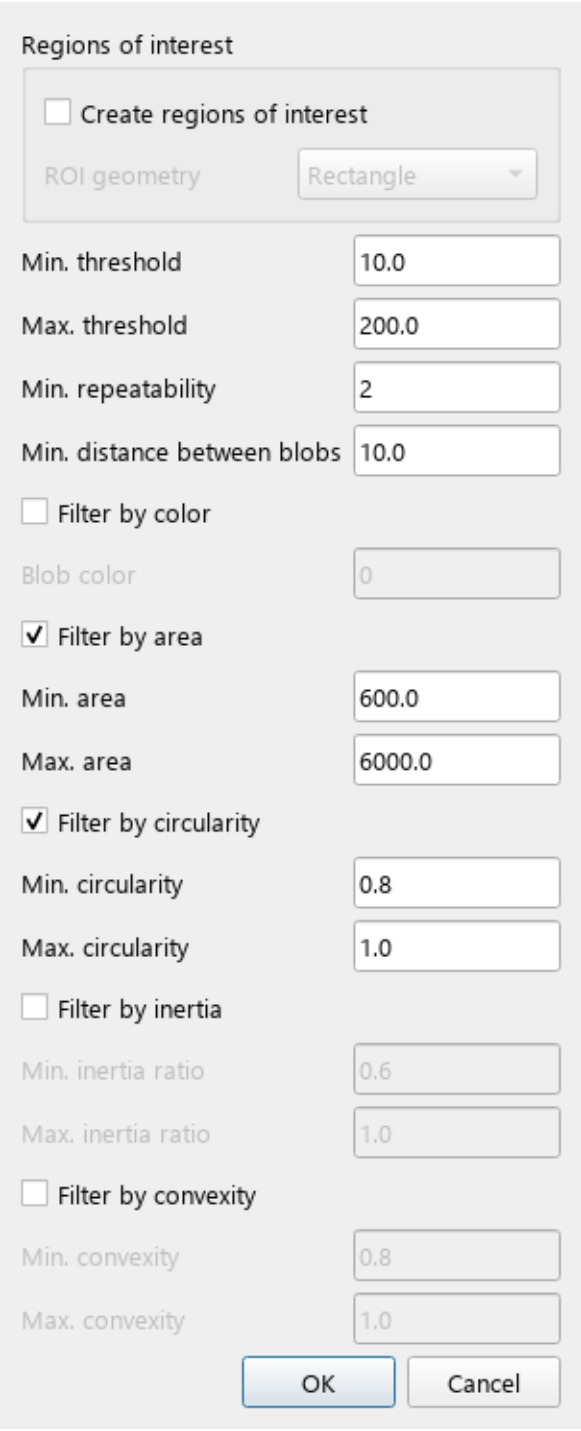
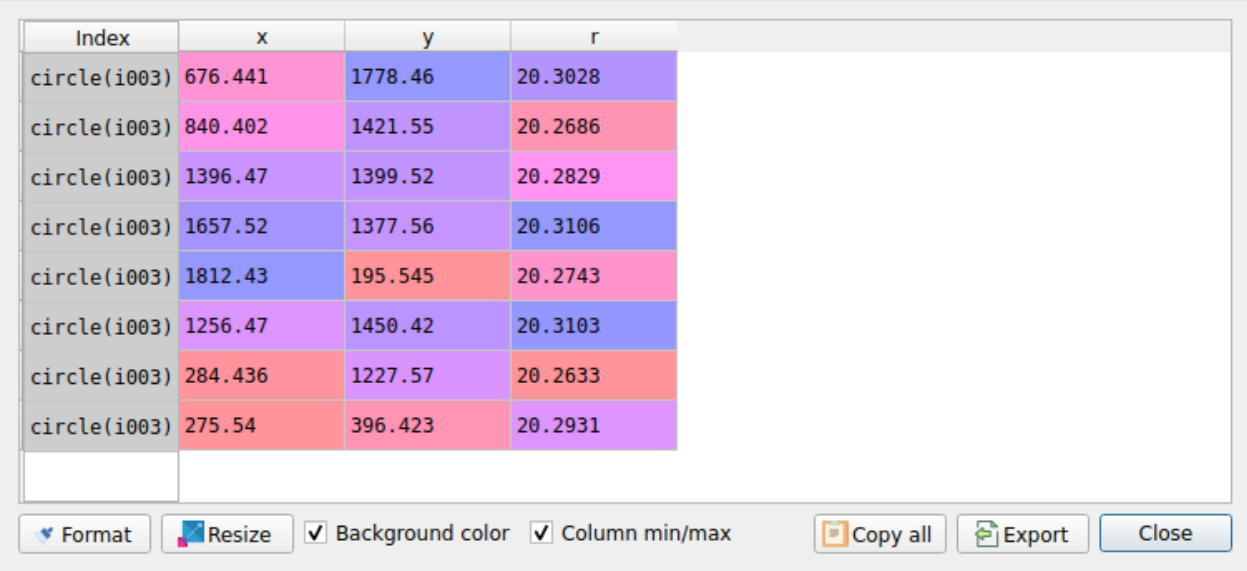


Fig. 43: The "Blob detection (OpenCV)" dialog opens. Set the parameters as shown on the screenshot, and click on "OK".



Index	x	y	r
circle(i003)	676.441	1778.46	20.3028
circle(i003)	840.402	1421.55	20.2686
circle(i003)	1396.47	1399.52	20.2829
circle(i003)	1657.52	1377.56	20.3106
circle(i003)	1812.43	195.545	20.2743
circle(i003)	1256.47	1450.42	20.3103
circle(i003)	284.436	1227.57	20.2633
circle(i003)	275.54	396.423	20.2931

Fig. 44: The “Results” dialog opens, showing the detected blobs: one line per blob, with the blob coordinates and radius.

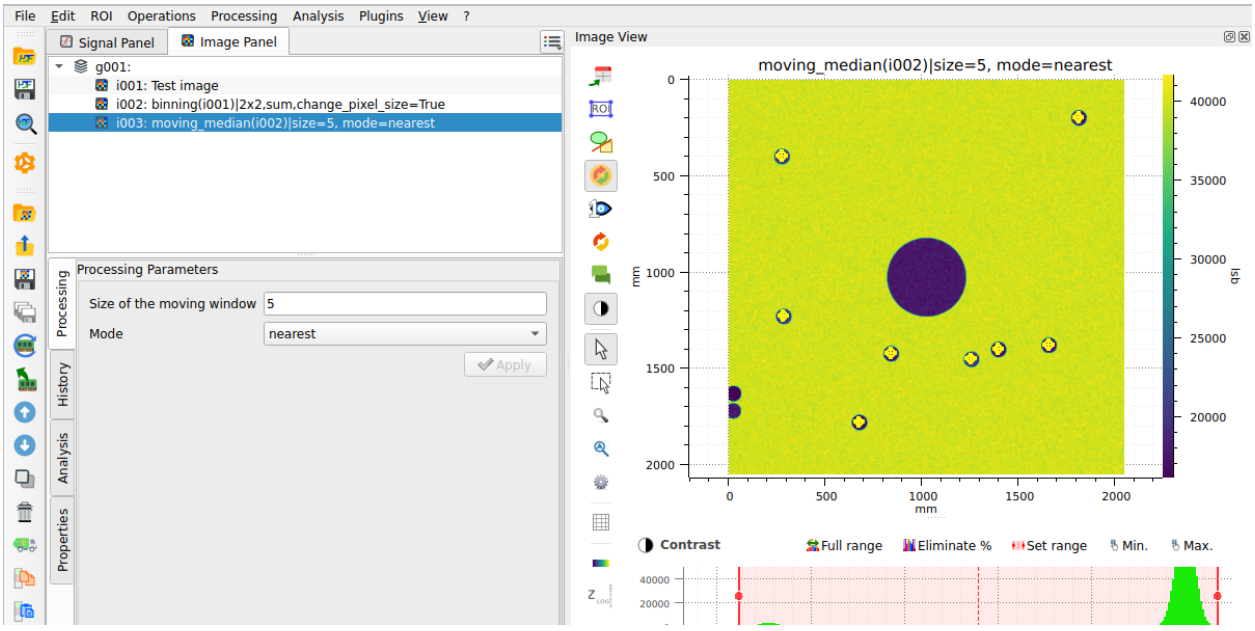


Fig. 45: The detected blobs are also added to the image metadata, and can be seen in the visualization panel on the right.

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.

Analyzing Fabry-Perot interferograms: ring detection and circle fitting

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>/_internal/datalab/data/tutorials/fabry_perot/` for the stand-alone version of DataLab).
>
> 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.

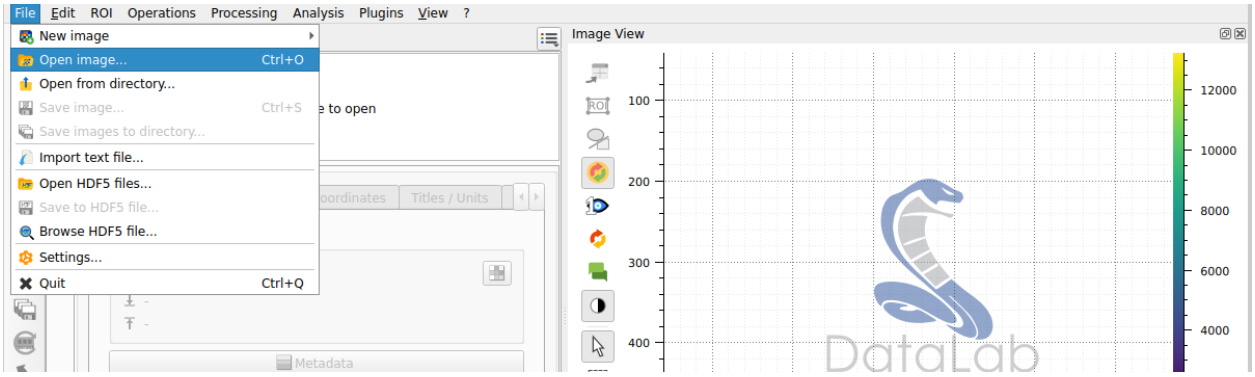


Fig. 46: Open the image files with "File > Open image…", or with the button in the toolbar, or by dragging and dropping the files into DataLab (on the panel on the right).

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 mouse.

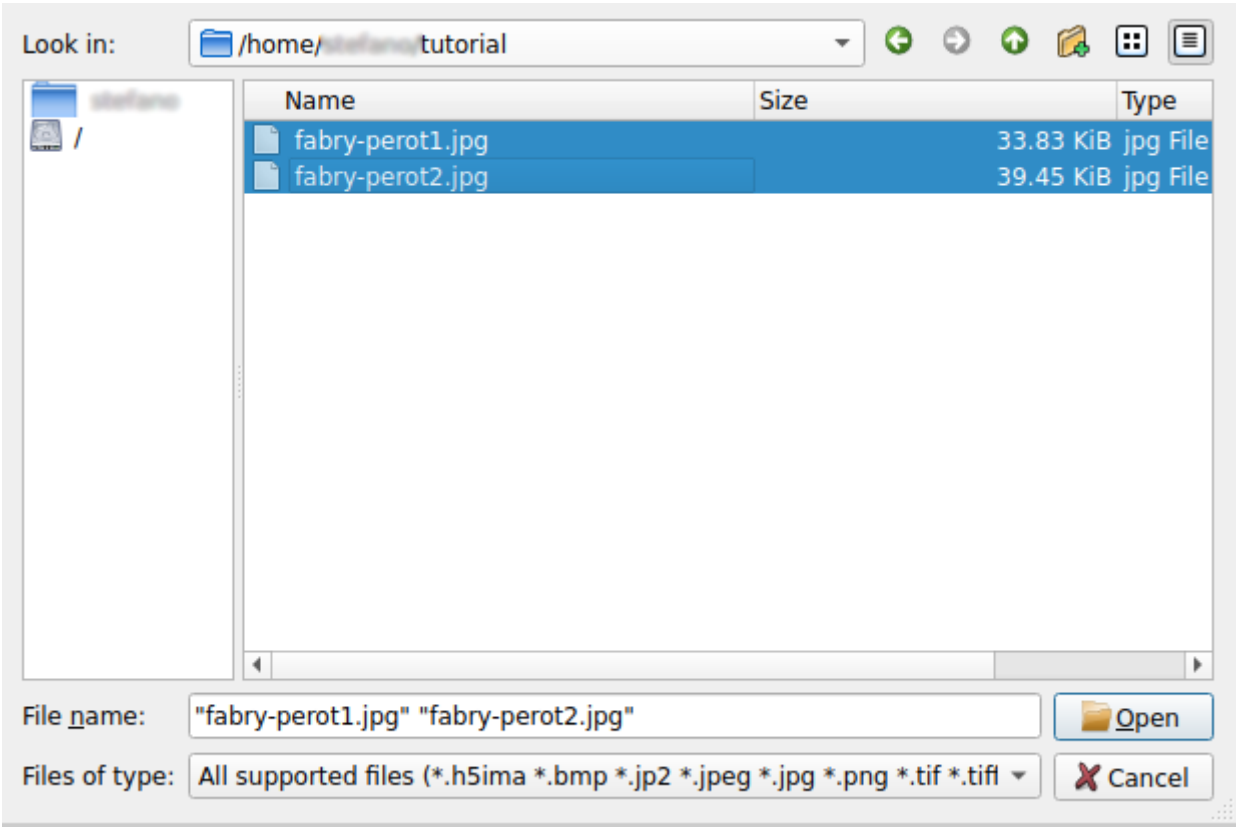


Fig. 47: Go to the tutorial data folder in DataLab's installation directory (or the folder where you placed the images), select "fabry-perot1.jpg" and "fabry-perot2.jpg" and click "Open".

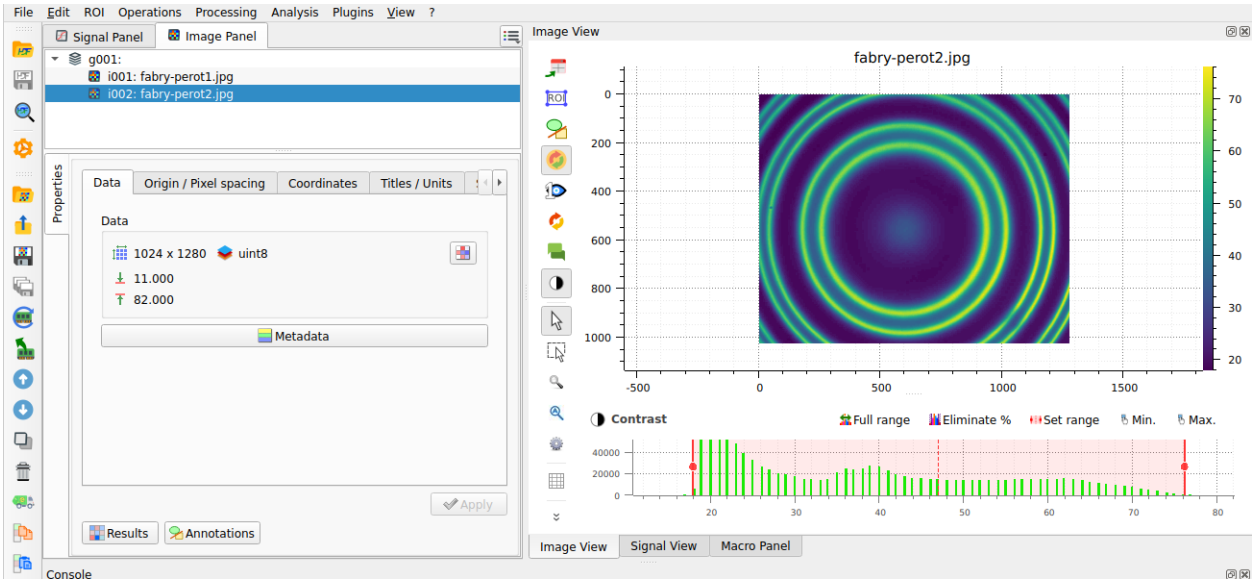


Fig. 48: Images loaded in DataLab. Zoom in and out with the right mouse button, pan the image with the middle mouse button.

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 "File > Settings…" in the menu, or the button in the toolbar.

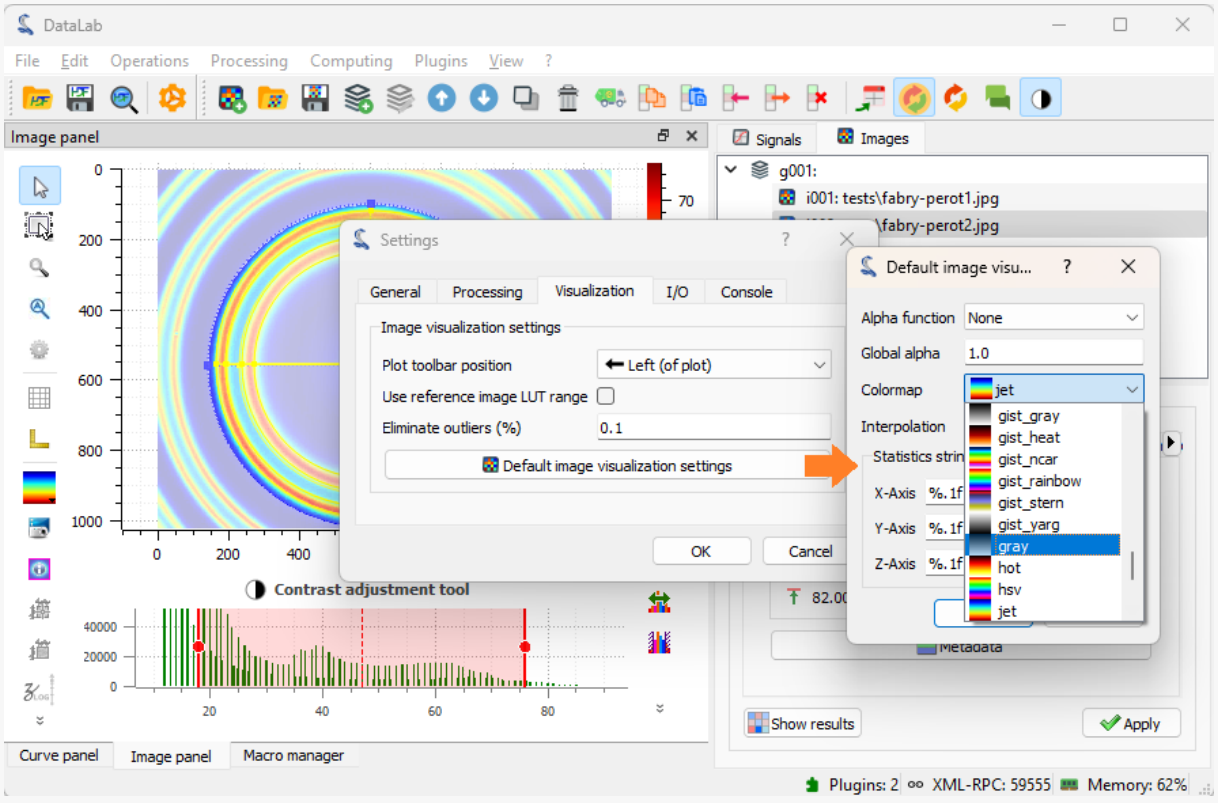


Fig. 49: Select the "Visualization" tab, and select the "gray" colormap.

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.

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).

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.

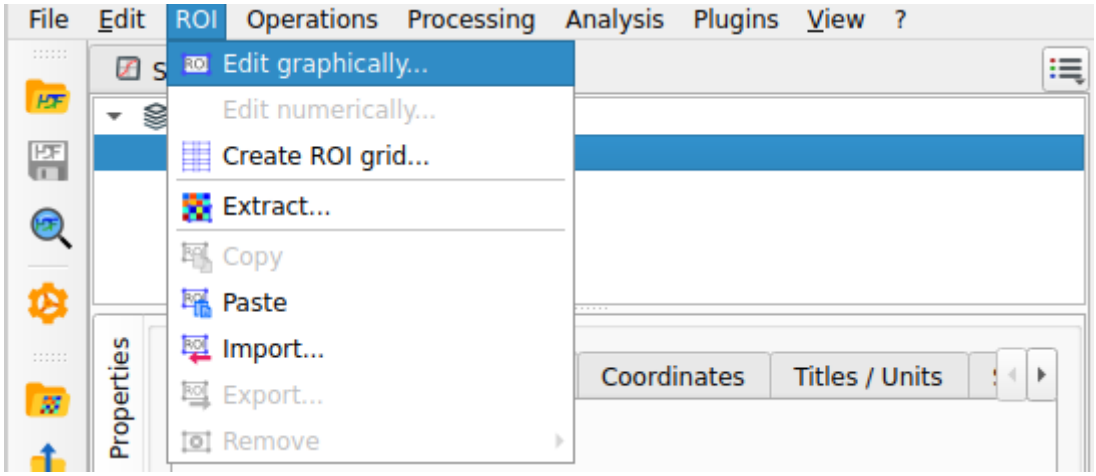


Fig. 50: Select the "Edit graphically" tool in the "ROI" menu.

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, so we leave the "Inverse ROI logic" 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.

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 "Close" 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…" entry in the "Analysis" menu.

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. To add annotations (cursors, labels, shapes, etc.) to the image or signal, click on the "Annotations" button of this window to show the annotation tools, or open the window directly with the "Edit annotations…" entry of the "View" menu: the annotations are stored with the image or signal (separately from its metadata), 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 to clean up the image. Select the "Delete object metadata" entry in the "Edit > Metadata" 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. This does not delete the annotations: to delete them, select the "Delete annotations" entry in the "Edit > Annotations" menu.

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

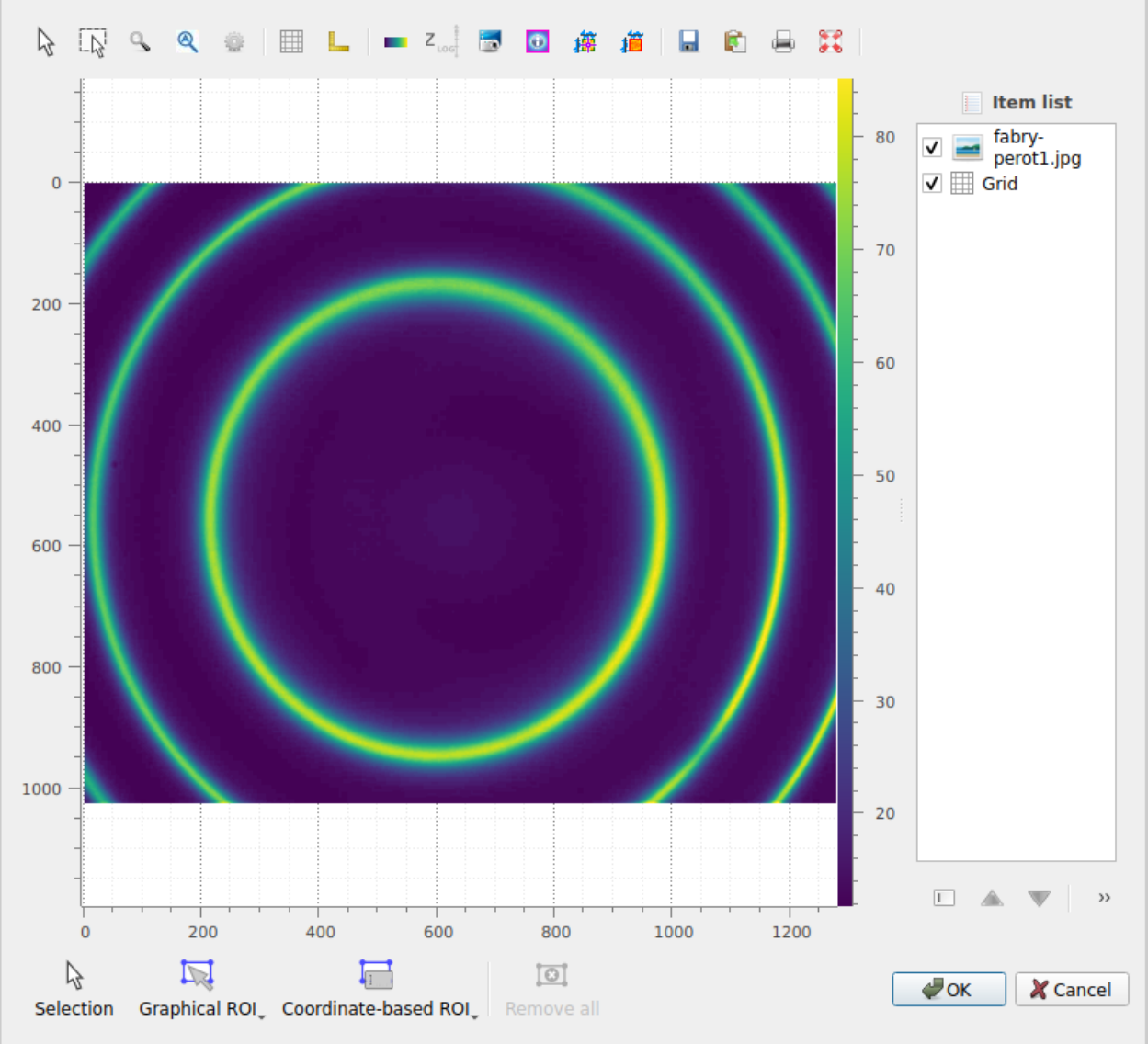


Fig. 51: The “Regions of interest” dialog opens. Click “Graphical ROI” and select a circular ROI. Resize the predefined ROI by dragging the handles. Note that you may change the ROI radius while keeping its center fixed by pressing the “Ctrl” key. Click “OK” to close the dialog.

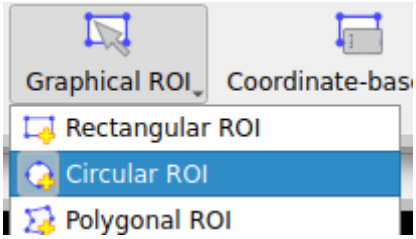


Fig. 52: The option to graphically define a circular ROI.

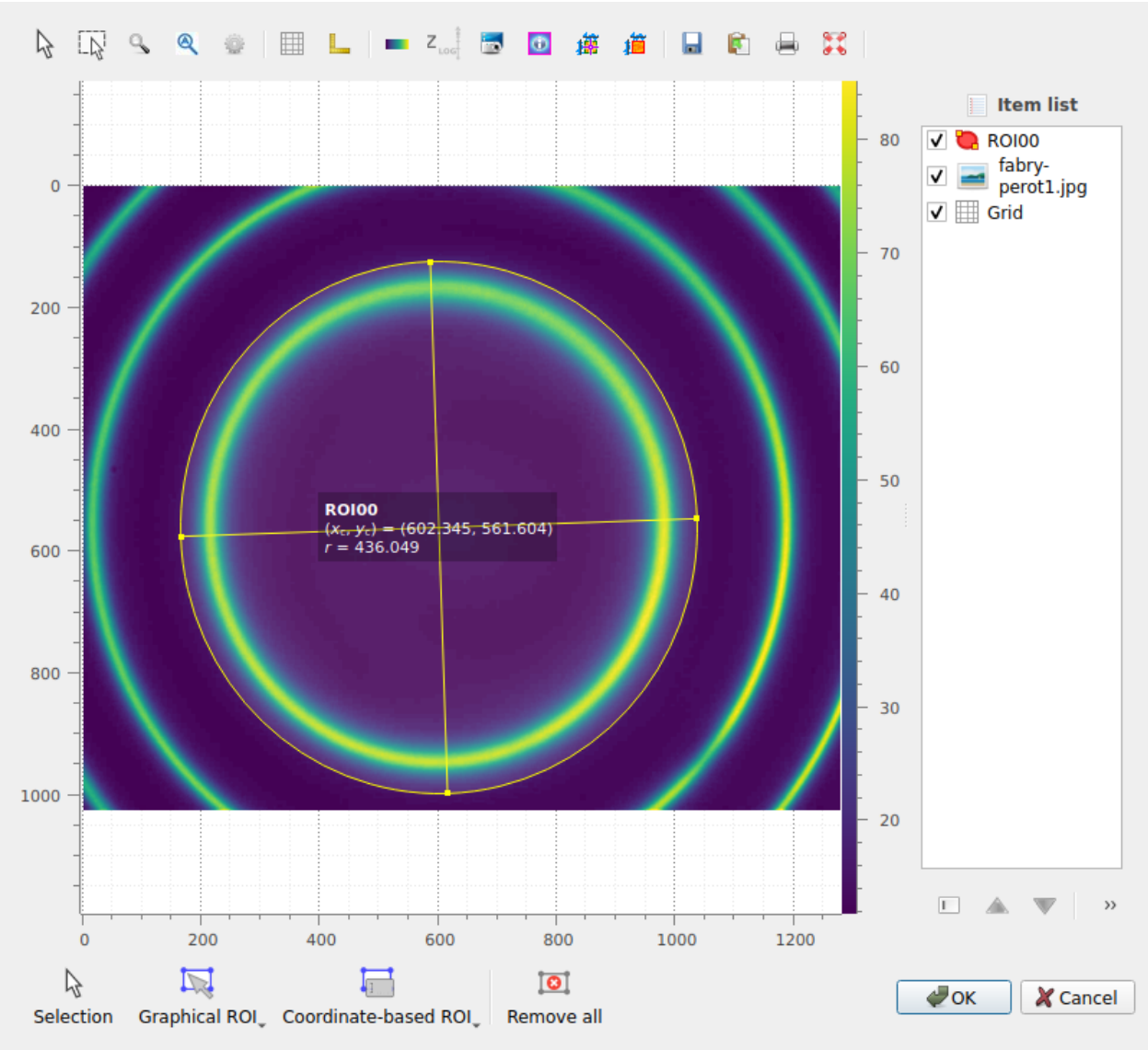


Fig. 53: The circular ROI created appears in the image.

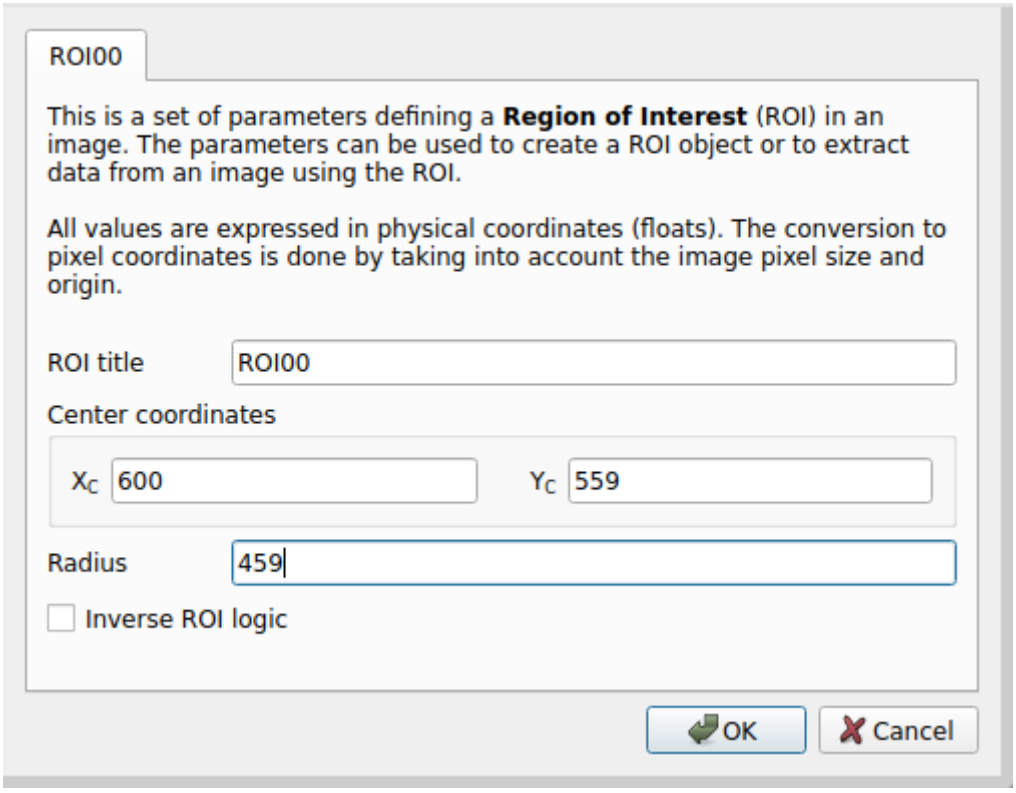


Fig. 54: The confirmation dialog.

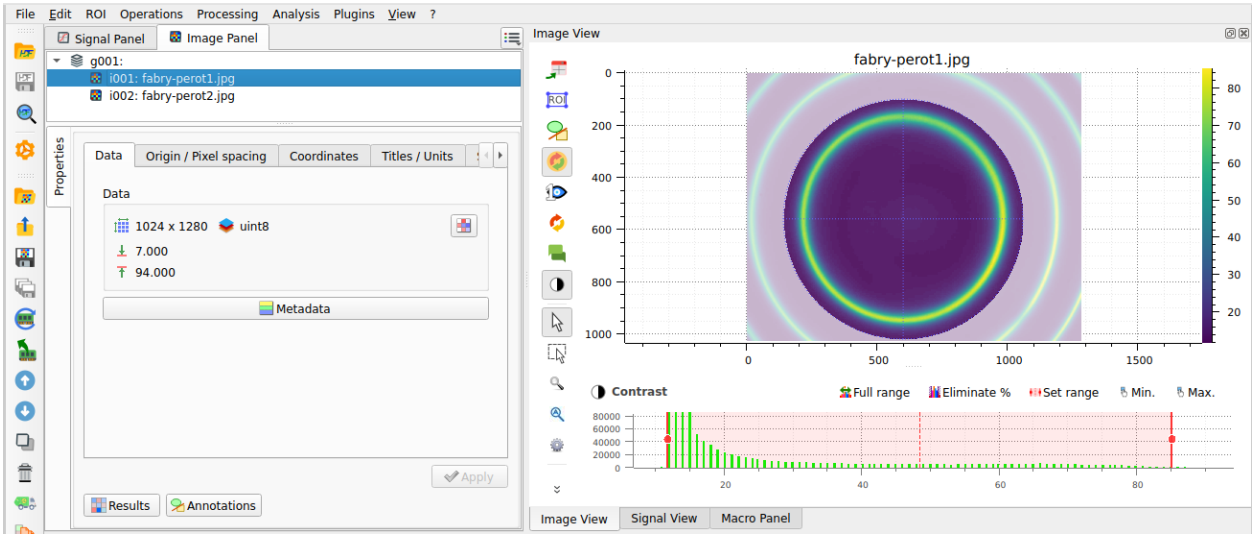


Fig. 55: The ROI displayed on the image.

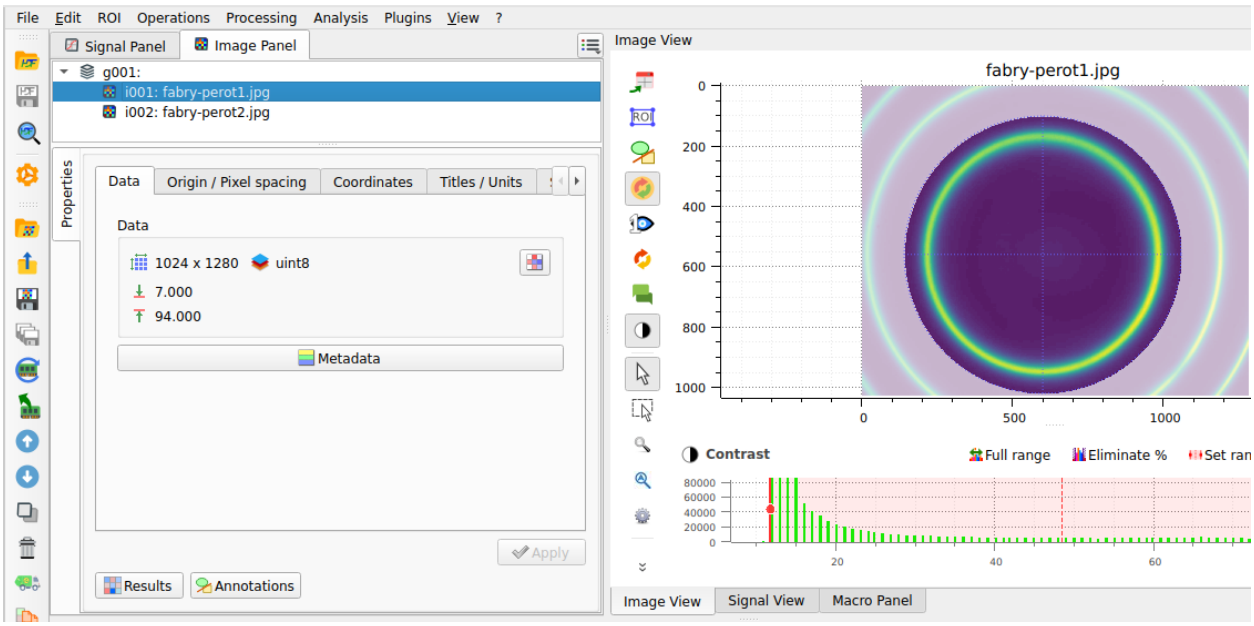


Fig. 56: The “Contour detection” tool in the “Analysis” menu.

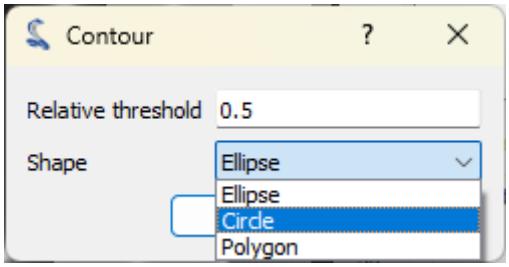


Fig. 57: The “Contour” parameters dialog.

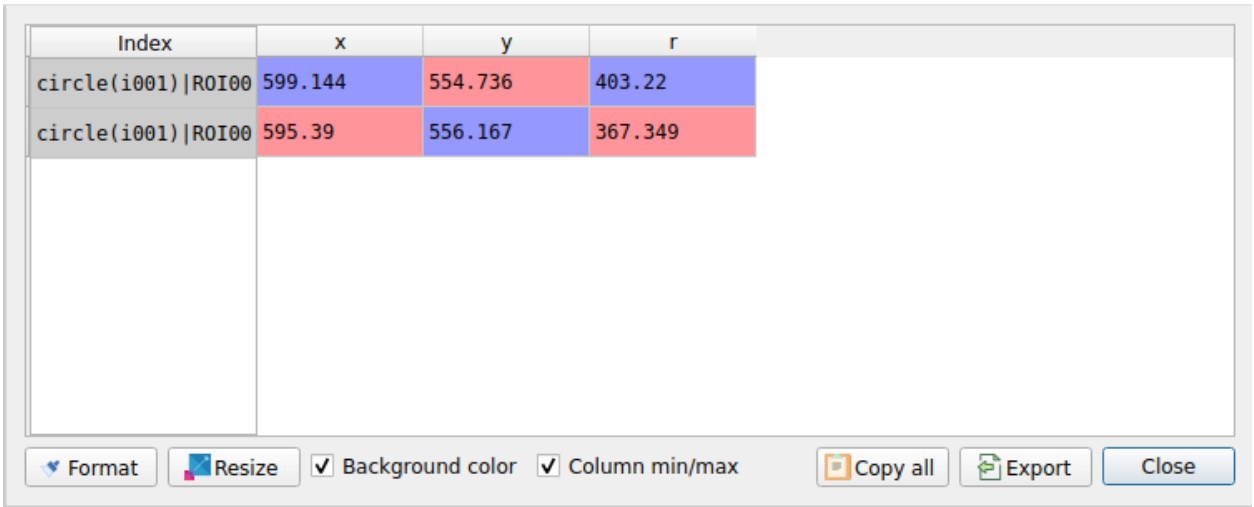


Fig. 58: The “Results” dialog.

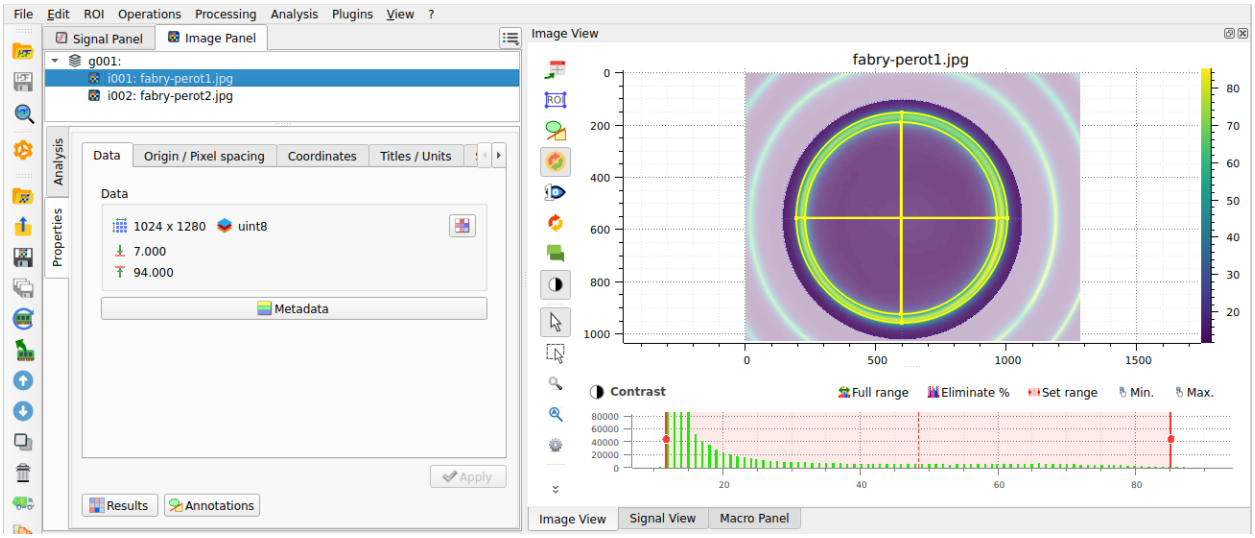


Fig. 59: The fitted circles are displayed on the image.

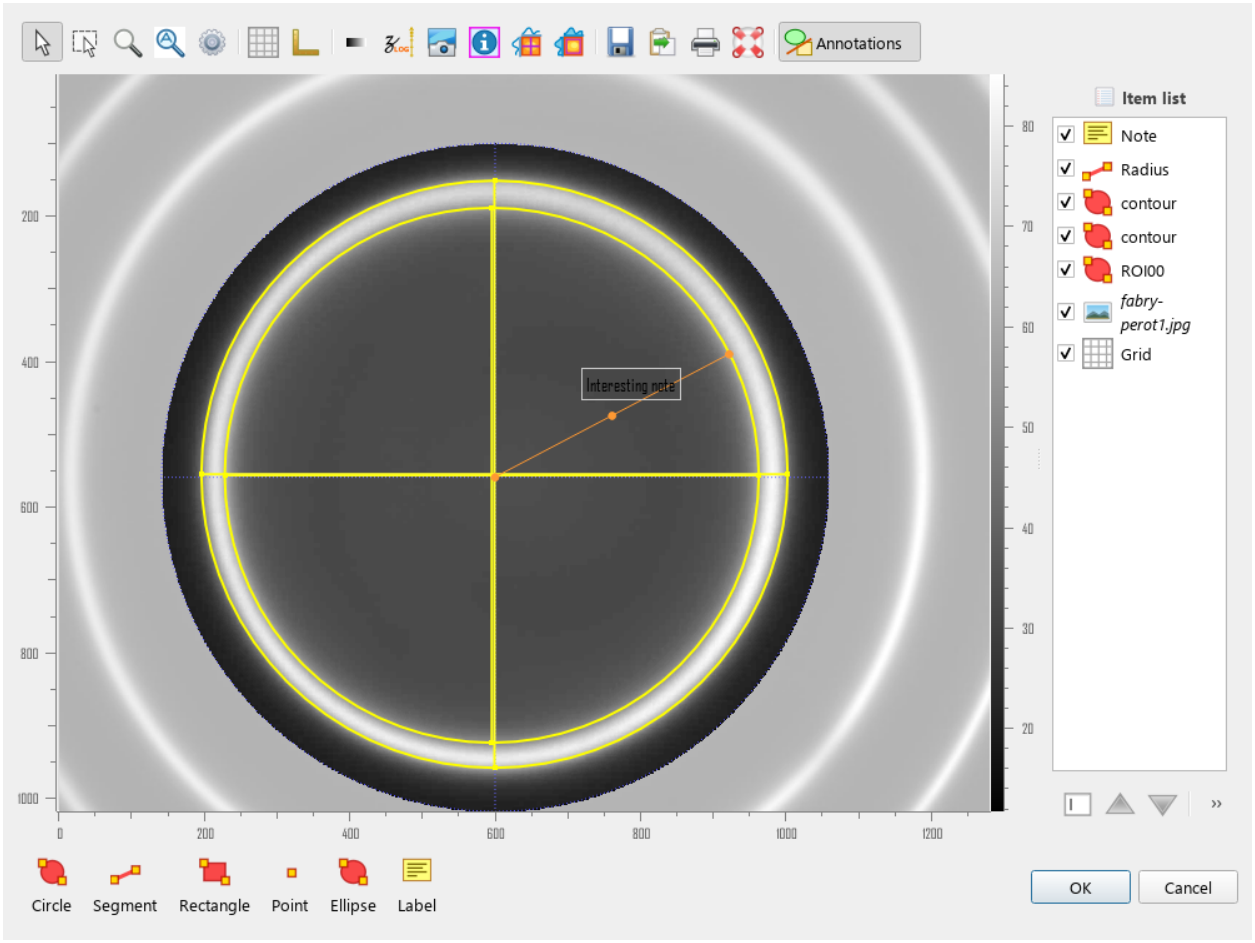


Fig. 60: The image displayed in a separate window. The ROI and the fitted circles are also displayed. Annotations can be added to the image with the annotation tools shown at the bottom of the window by the “Annotations” button. The annotations are saved together with the image data when the workspace is saved. Click on “OK” to close the window.

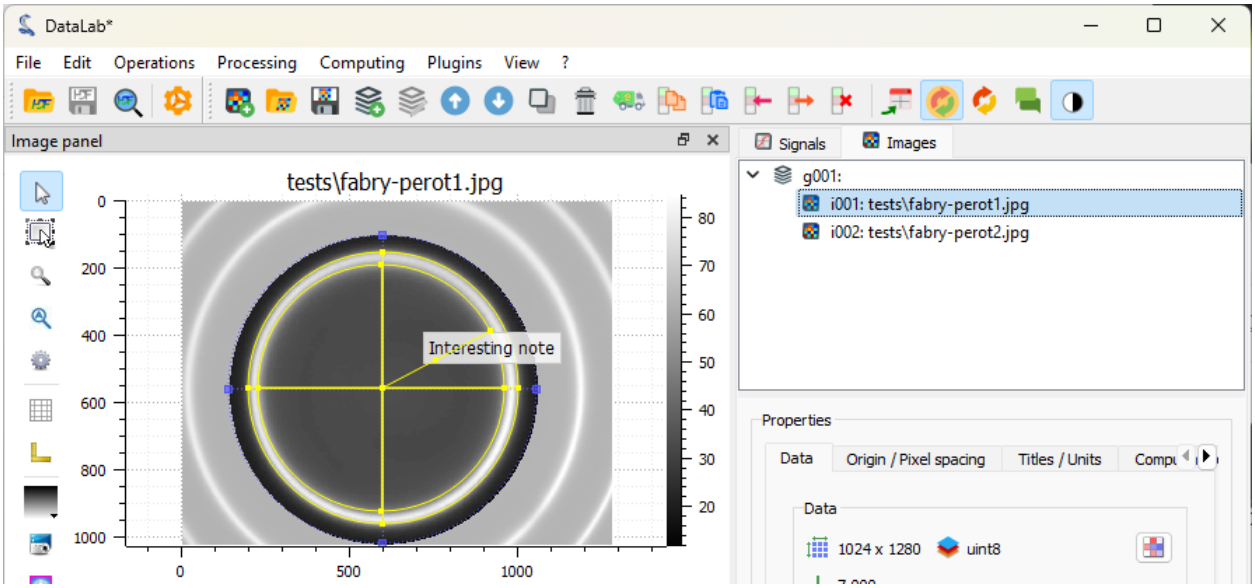


Fig. 61: The image is displayed in the main window, together with the annotations.

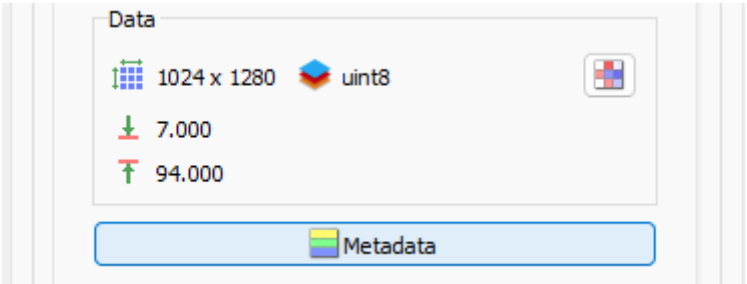
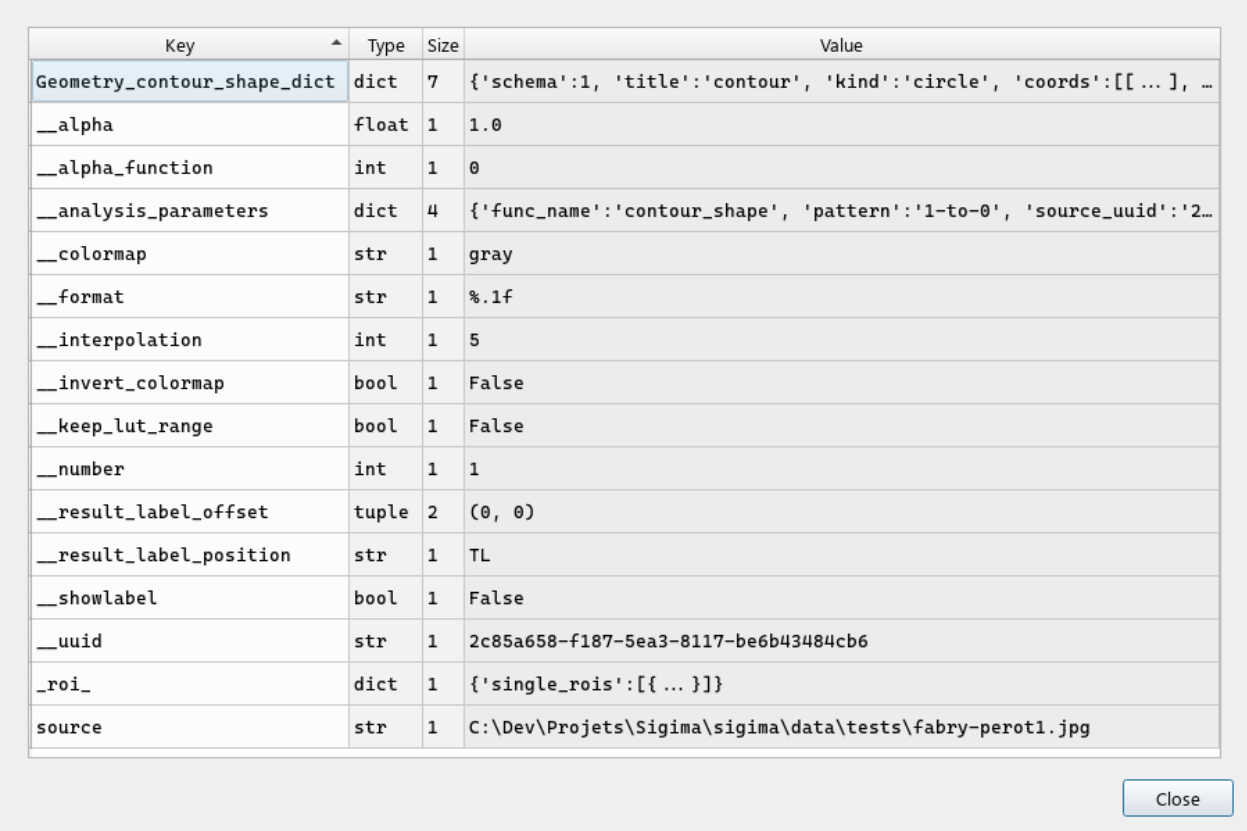


Fig. 62: The “Metadata” button is located below the image list.



Key	Type	Size	Value
Geometry_contour_shape_dict	dict	7	{'schema':1, 'title':'contour', 'kind':'circle', 'coords':[[...], …
__alpha	float	1	1.0
__alpha_function	int	1	0
__analysis_parameters	dict	4	{'func_name':'contour_shape', 'pattern':'1-to-0', 'source_uuid':'2…
__colormap	str	1	gray
__format	str	1	%.1f
__interpolation	int	1	5
__invert_colormap	bool	1	False
__keep_lut_range	bool	1	False
__number	int	1	1
__result_label_offset	tuple	2	(0, 0)
__result_label_position	str	1	TL
__showlabel	bool	1	False
__uuid	str	1	2c85a658-f187-5ea3-8117-be6b43484cb6
roi	dict	1	{'single_rois':[{ ... }]}
source	str	1	C:\Dev\Projets\Sigima\sigima\data\tests\fabry-perot1.jpg

Close

Fig. 63: The “Metadata” dialog opens. Among other information, it displays some style information (e.g. the colormap) and the ROI.

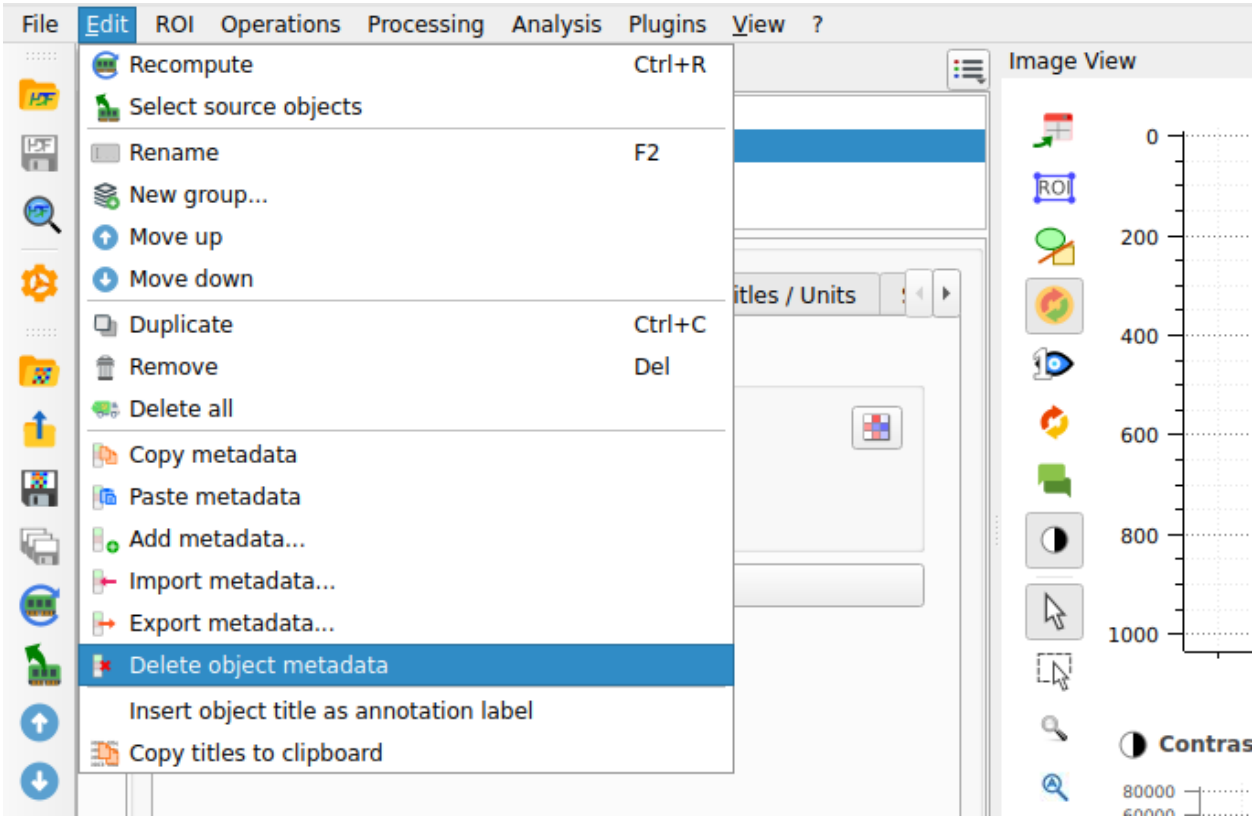


Fig. 64: Select the “Delete object metadata” entry in the “Edit > Metadata” menu.

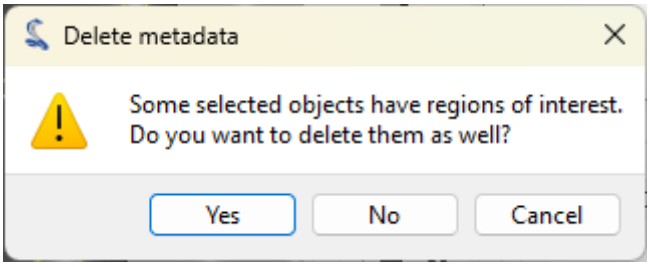


Fig. 65: The “Delete metadata” dialog. Click “No” to keep the ROI and delete the rest of the metadata.

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" 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.

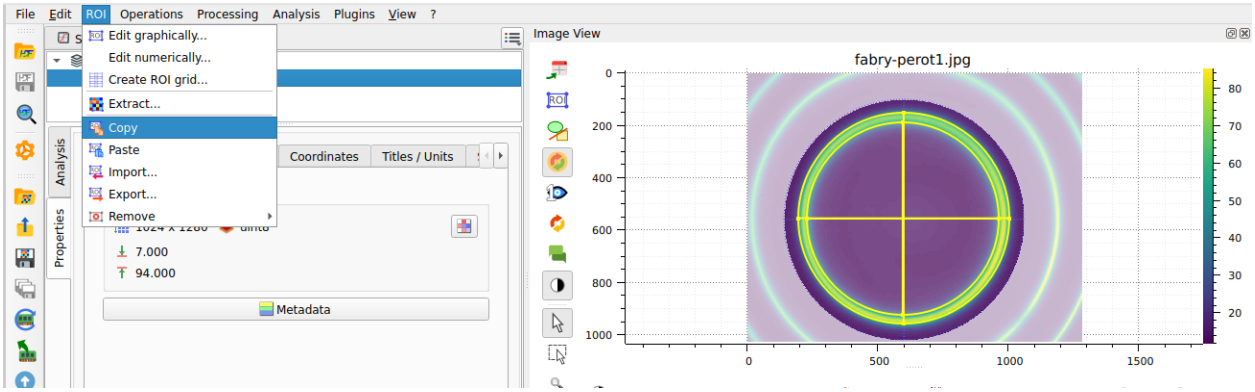


Fig. 66: The "Copy" entry in the "ROI" menu.

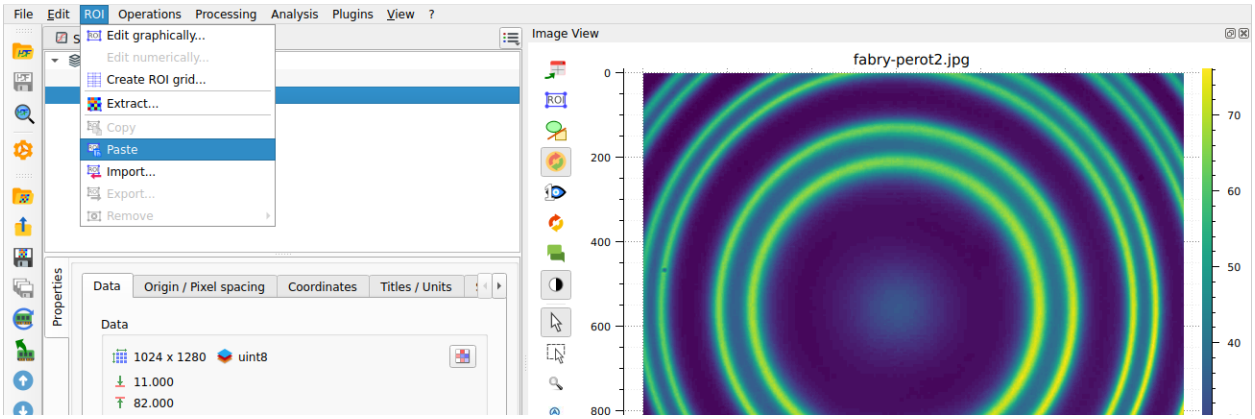


Fig. 67: The "Paste" entry in the "ROI" menu, or the  button in the toolbar.

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 "Close".

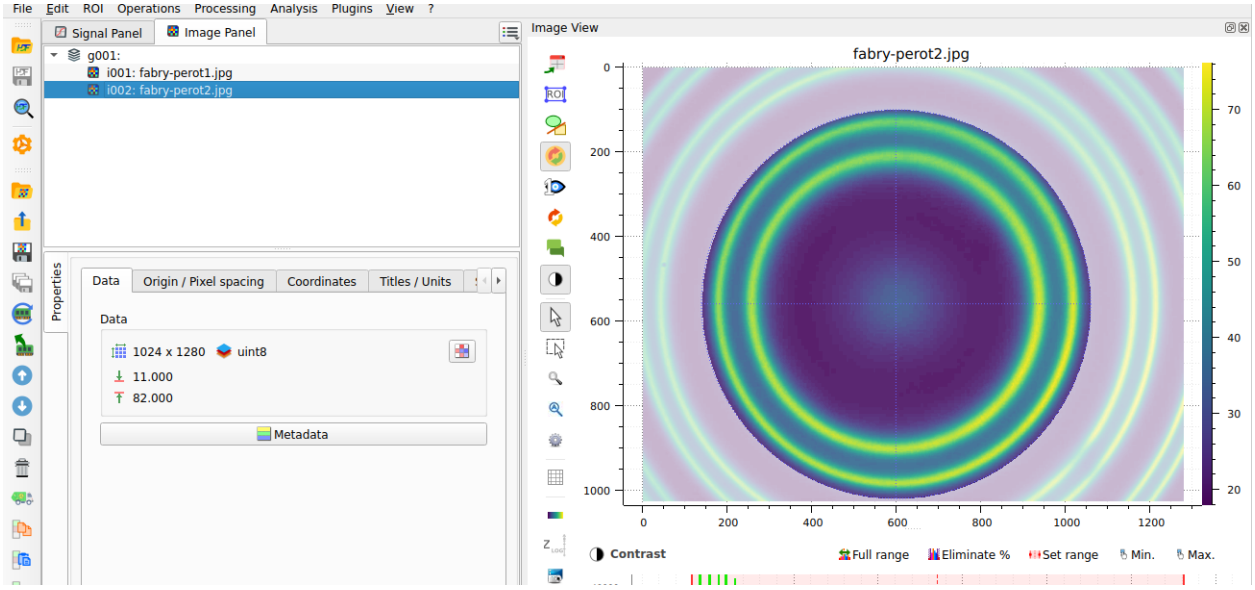


Fig. 68: The ROI is added to the second image.

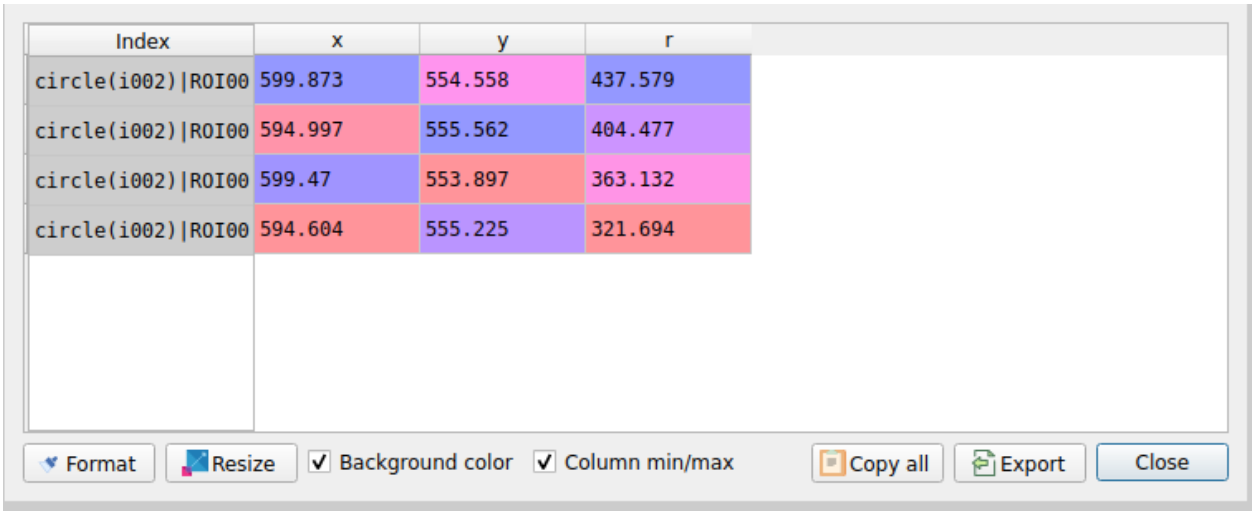


Fig. 69: The “Results” dialog for the second image.

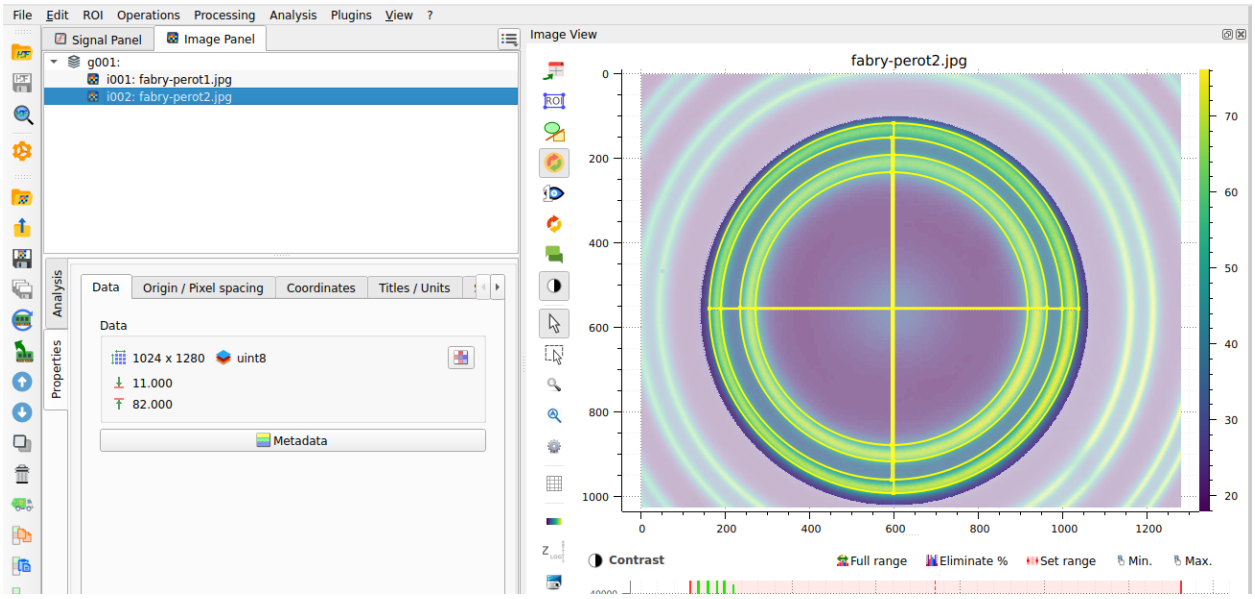


Fig. 70: The fitted circles are displayed on the image.

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 extraction…" 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 "Analysis > Intensity profiles" menu and the "Line 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 Y coordinate of the row (or the X coordinate of the column) directly by clicking on "Edit profile parameters" in the "Line profile" dialog.

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

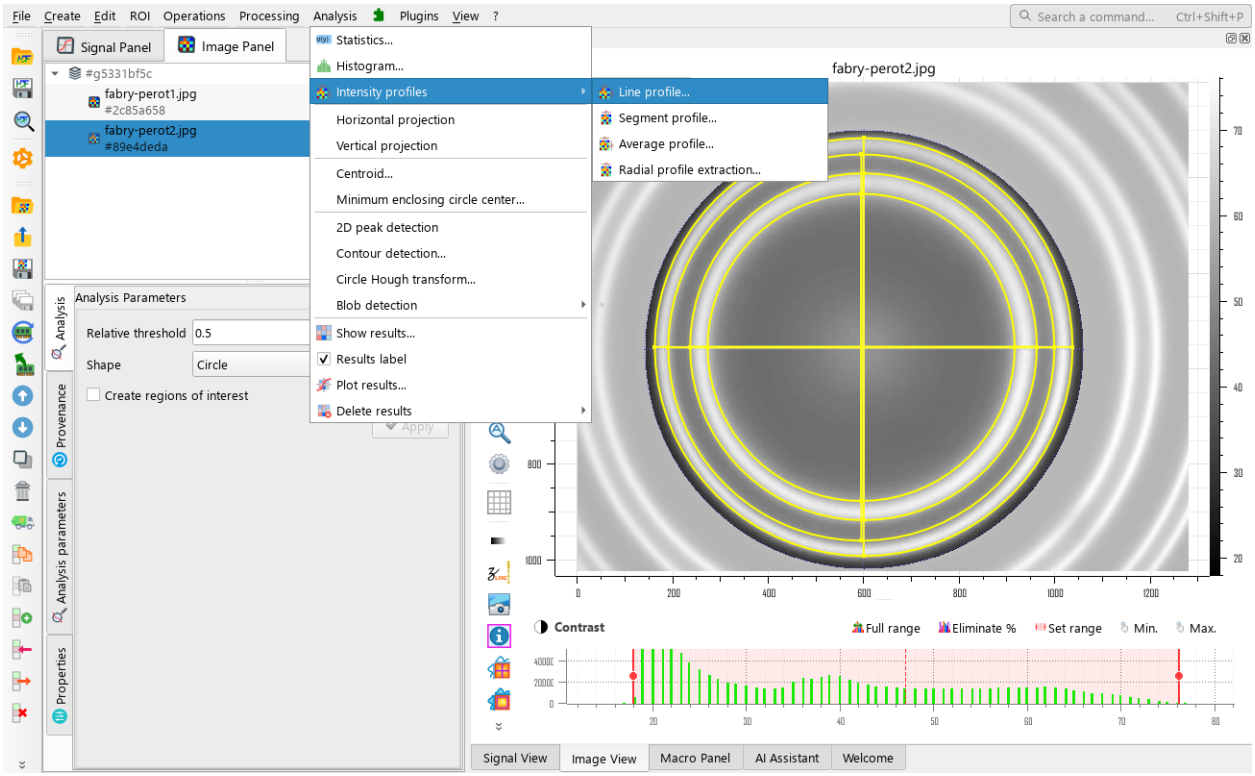


Fig. 71: Select the “Line profile…” entry in the “Analysis > Intensity profiles” menu.

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 files…” (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.

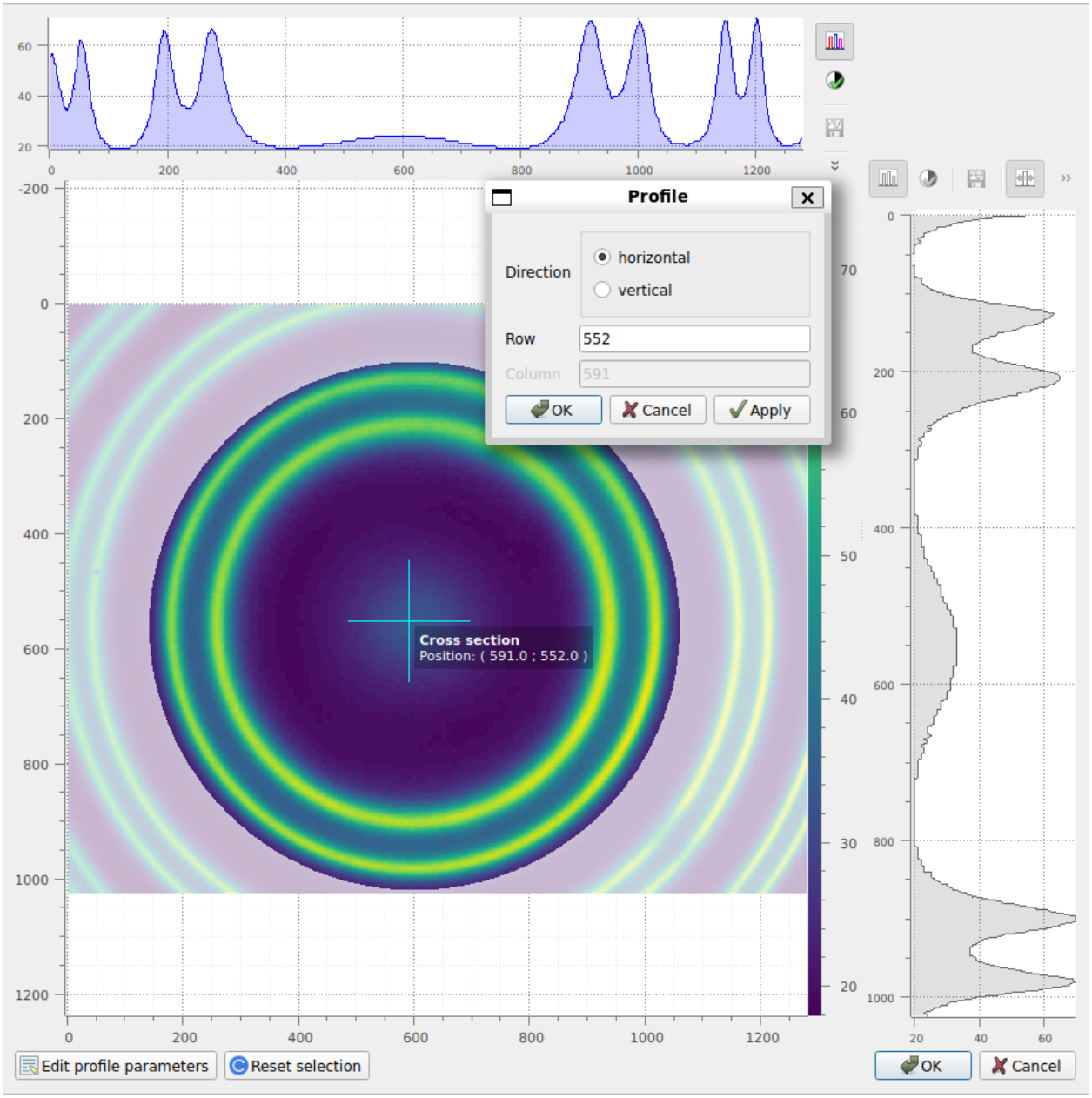


Fig. 72: The “Profile” dialog opens. Enter the Y coordinate of the horizontal profile (or the X coordinate of the vertical profile) in the dialog box that opens. Click “OK”.

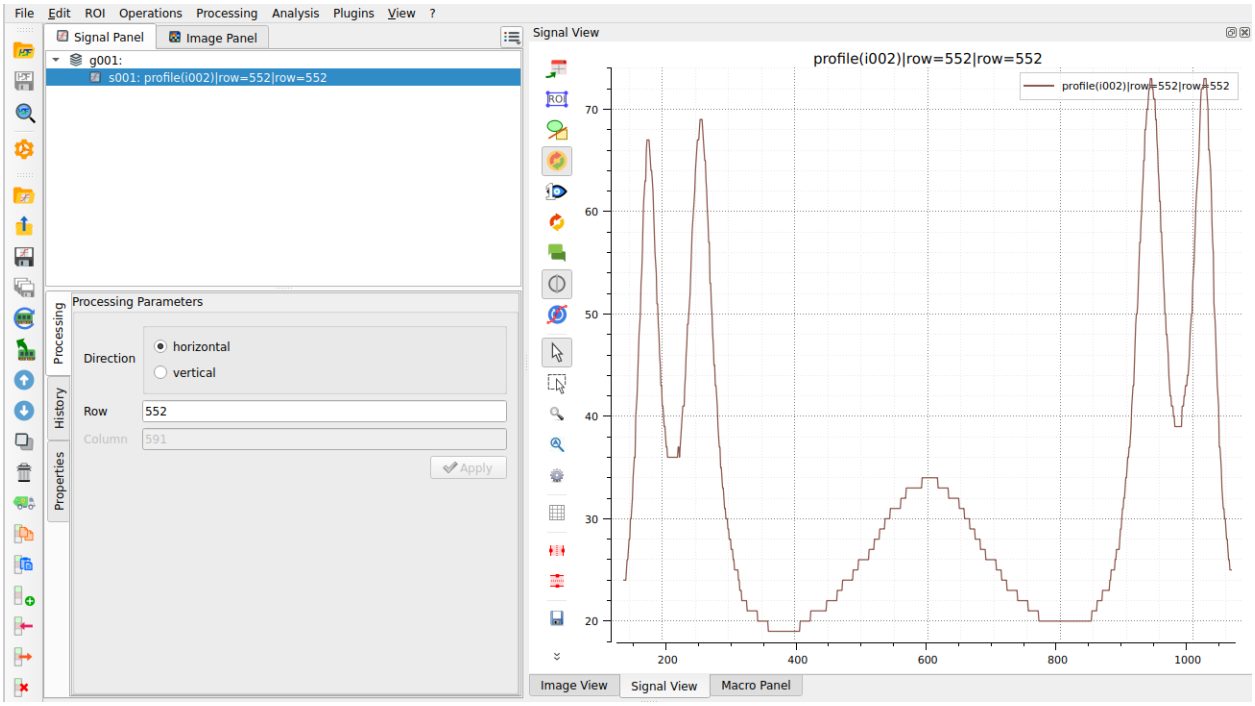


Fig. 73: The intensity profile is added to the “Signals” panel, and DataLab switches to this panel to display the profile.

Laser beam profiling: measuring beam size along the propagation axis

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>/_internal/datalab/data/tutorials/laser_beam/` for the stand-alone version of DataLab).
>
> Alternatively, they can be downloaded from the online documentation at https://datalab-platform.com.

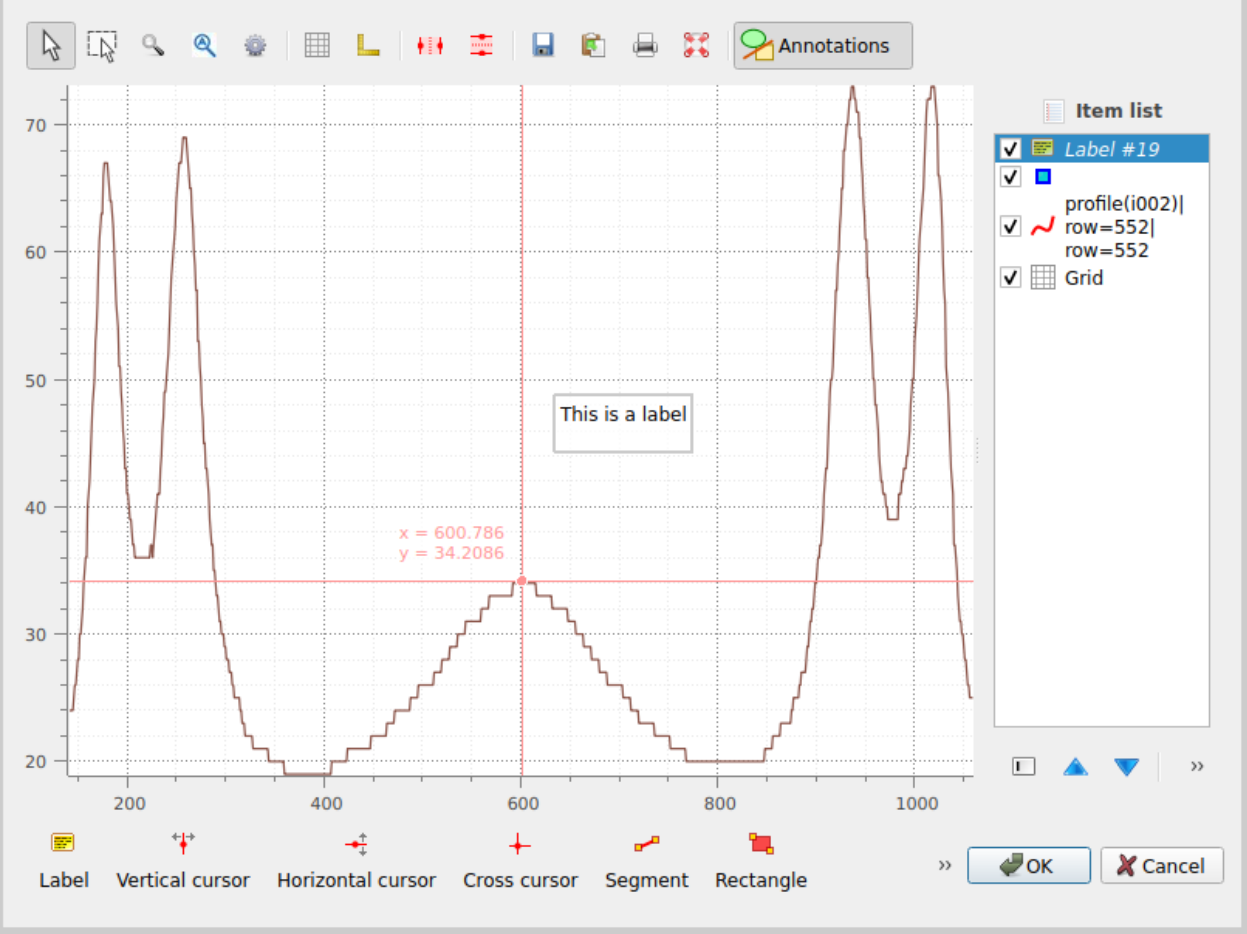


Fig. 74: The signal is displayed in a separate window. Here, we added vertical cursors and a text label. As for the images, the annotations are saved together with the signal data when the workspace is saved. Click on “OK” to close the window.

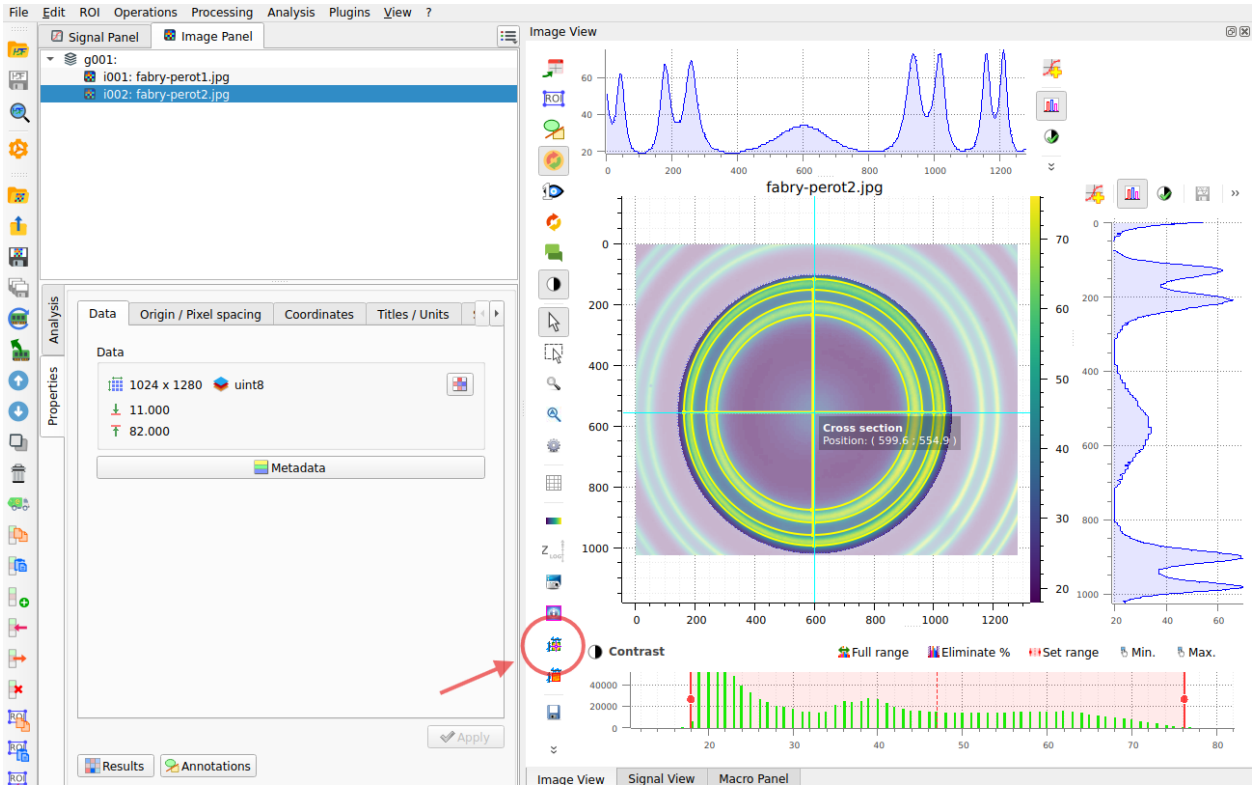


Fig. 75: The cross section tool in the view panel. In the red circle, the button to activate it. In the blue circle, the button to transfer the profile to the “Signals” panel.

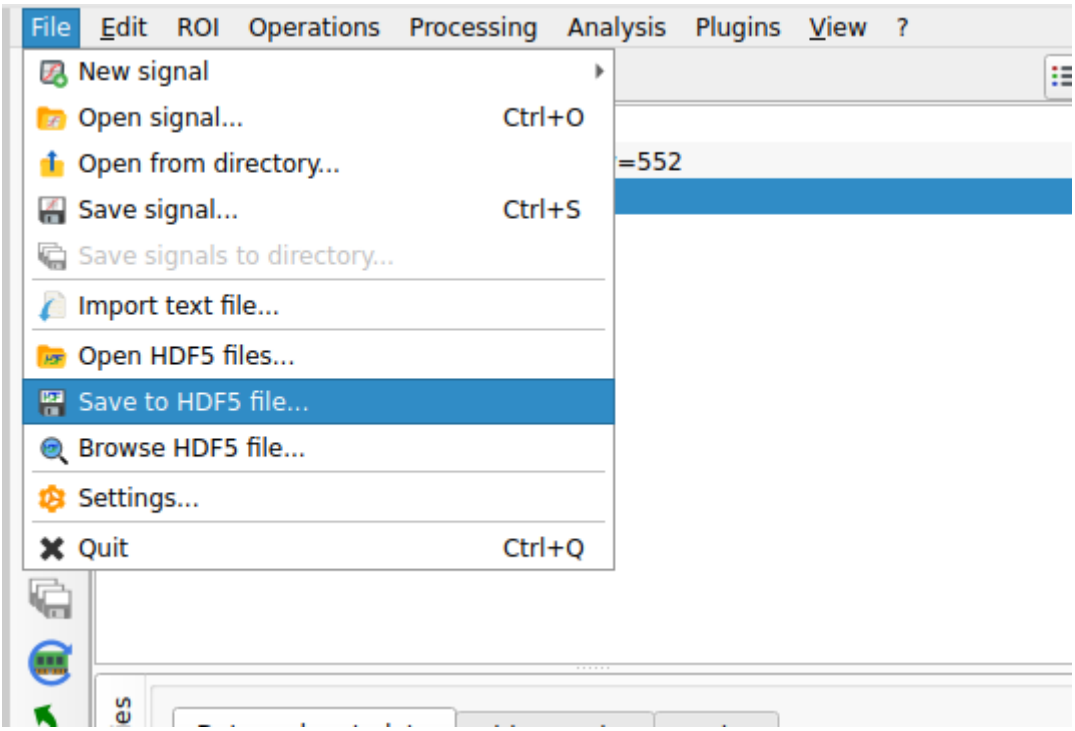


Fig. 76: Save the workspace to a file with “File > Save to HDF5 file…”, or the button in the toolbar.

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.

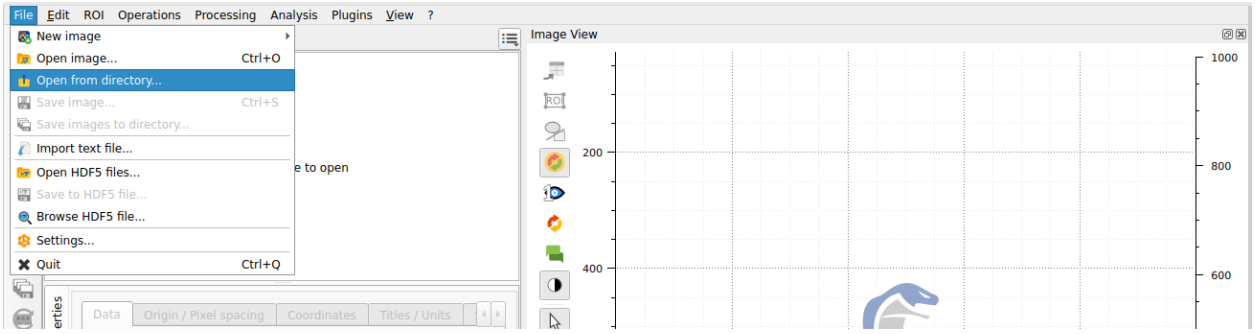


Fig. 77: The “File > Open from directory…” menu

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.

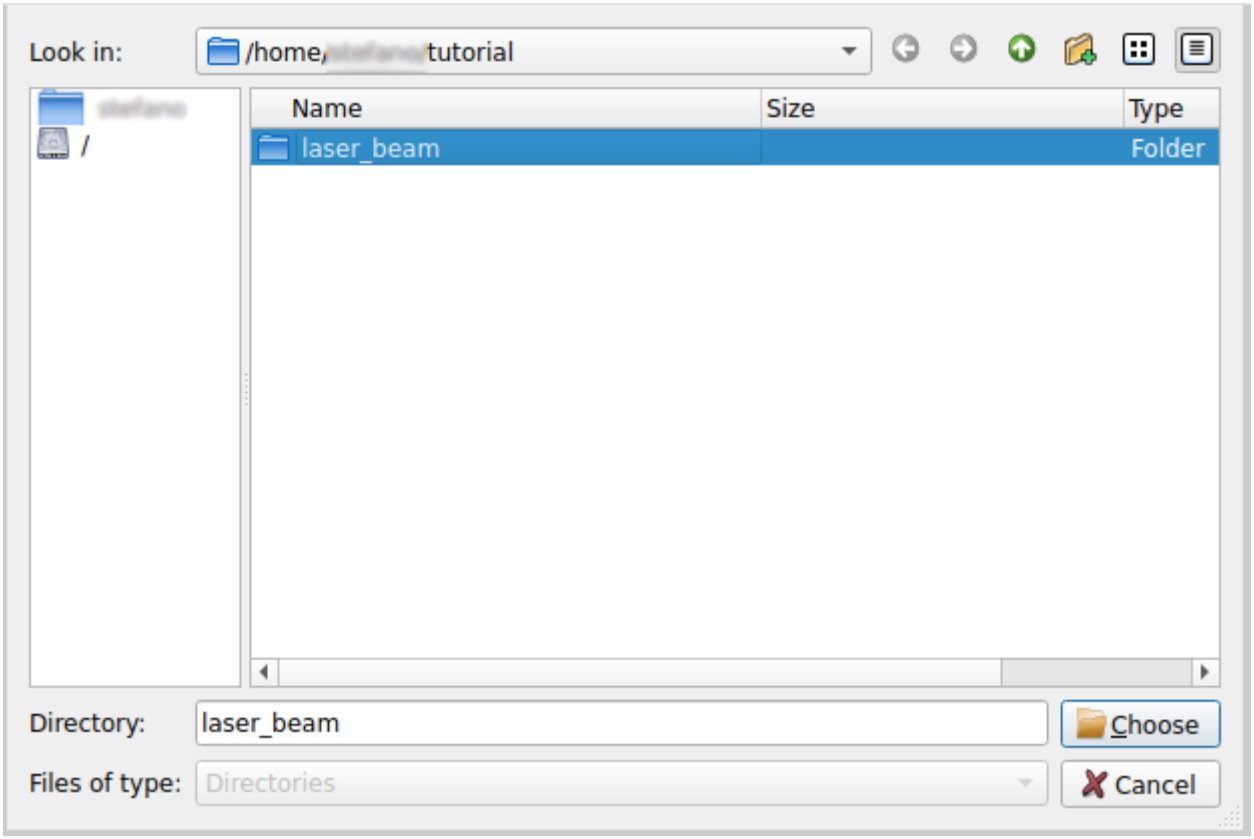


Fig. 78: Select the folder containing the images and click “Choose”.

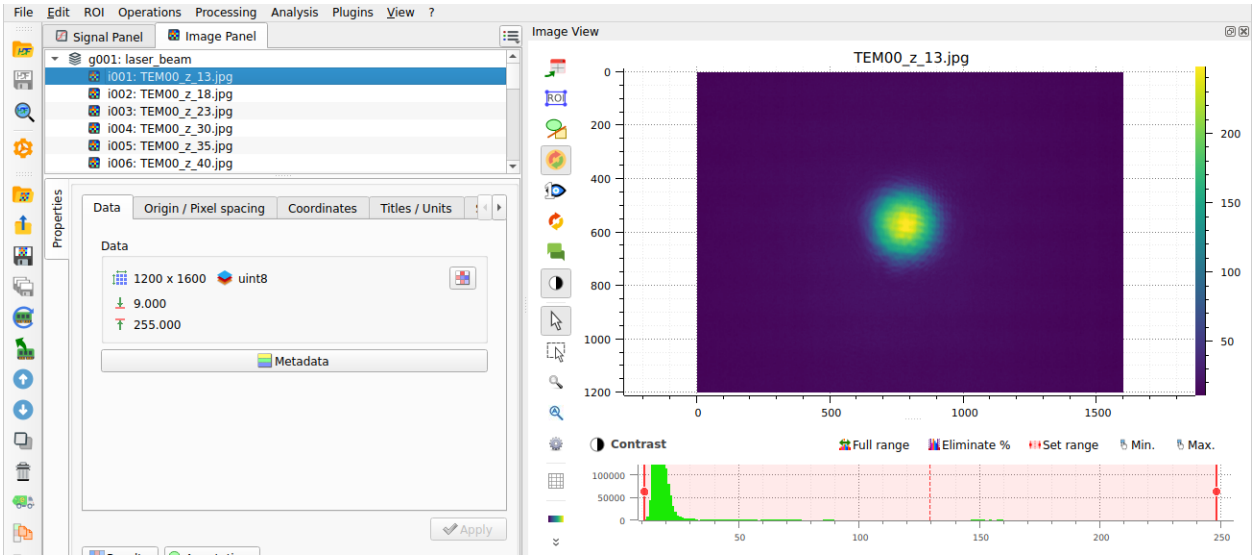


Fig. 79: Zoom in and out with the right mouse button. Pan the image with the middle mouse button.

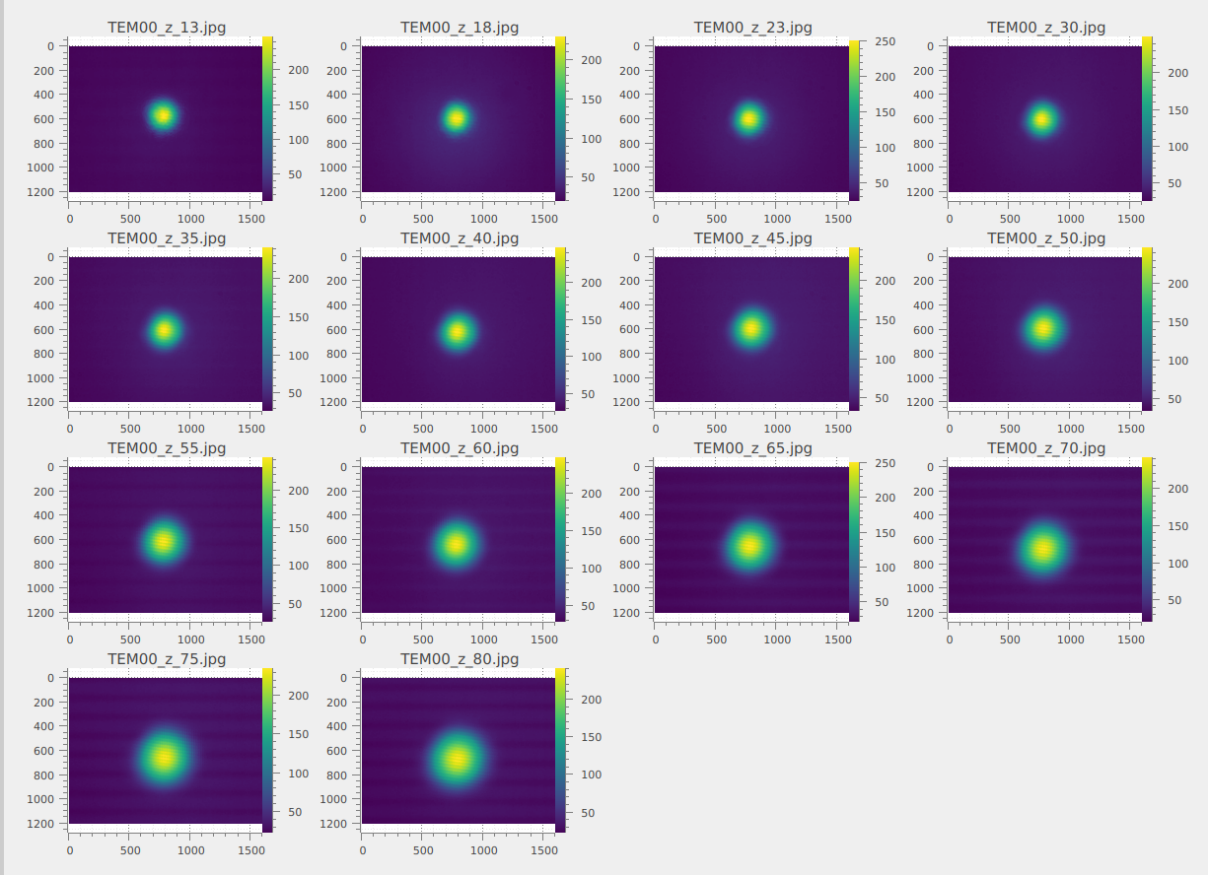
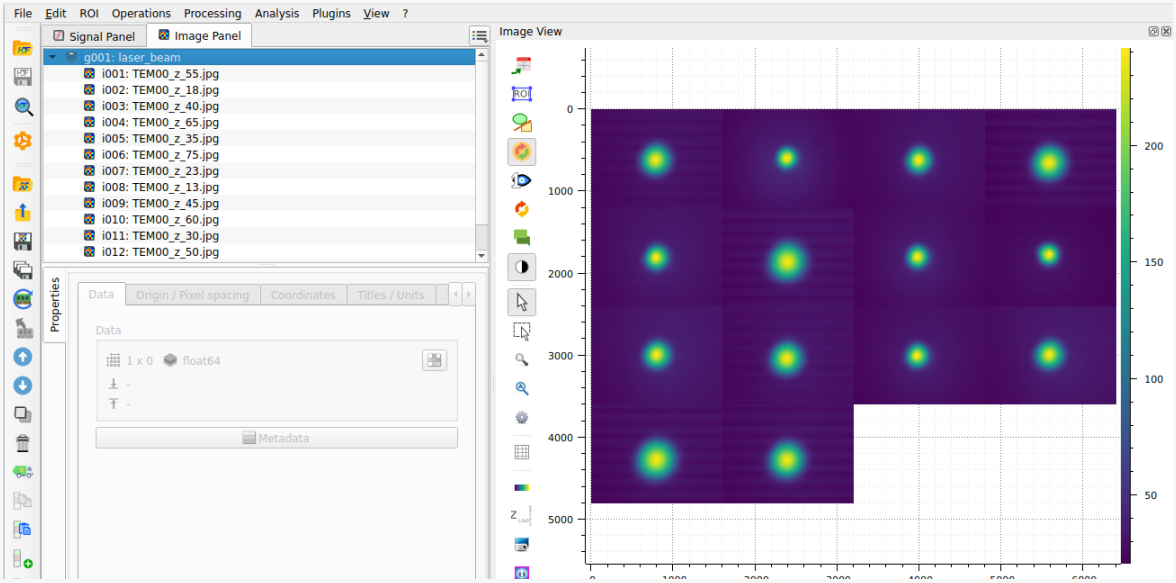


Fig. 80: Viewing images side by side.

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.
Both the “Distribute on a grid” and “Reset image positions” options modify the images in place, without creating new images. For this reason, these operations are not recorded in the History Panel.

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 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, 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 > Clip-

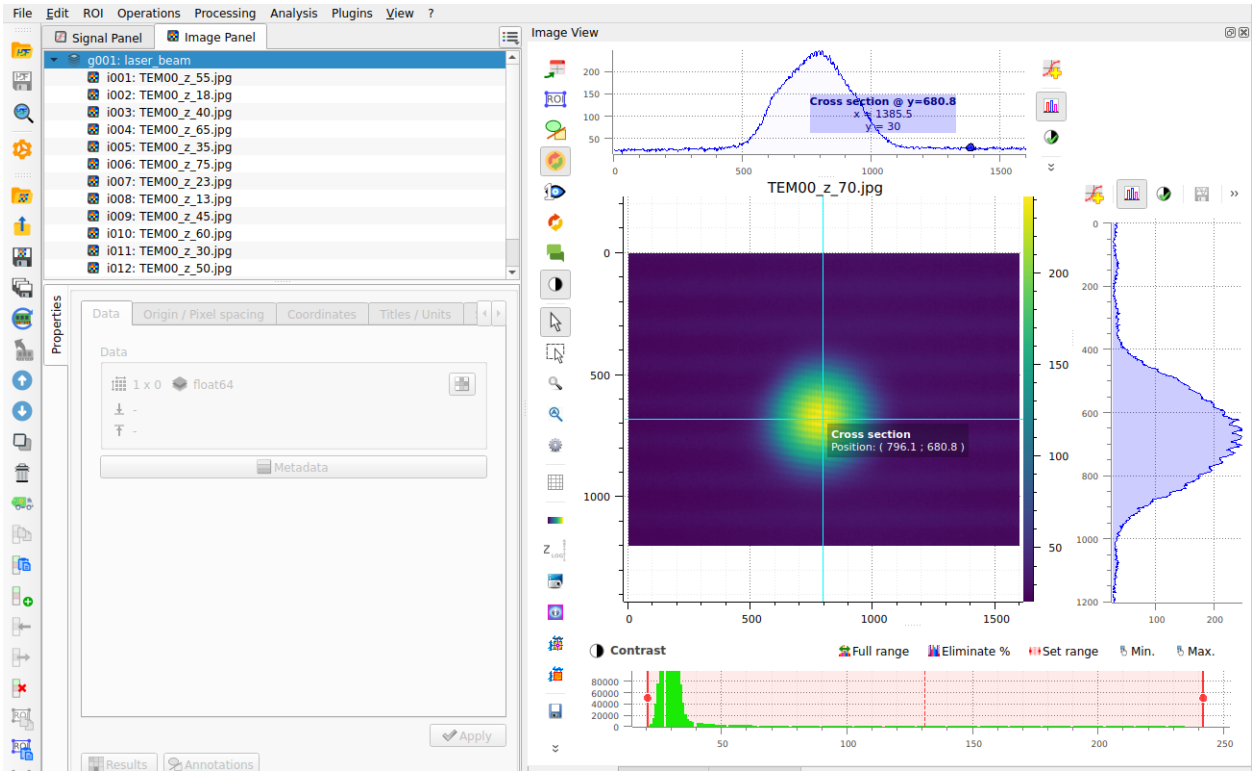


Fig. 81: An image from the “Image panel”. To display the curve marker, select the profile curve and right-click to open the context menu, then choose “Markers > Bound to active item”.

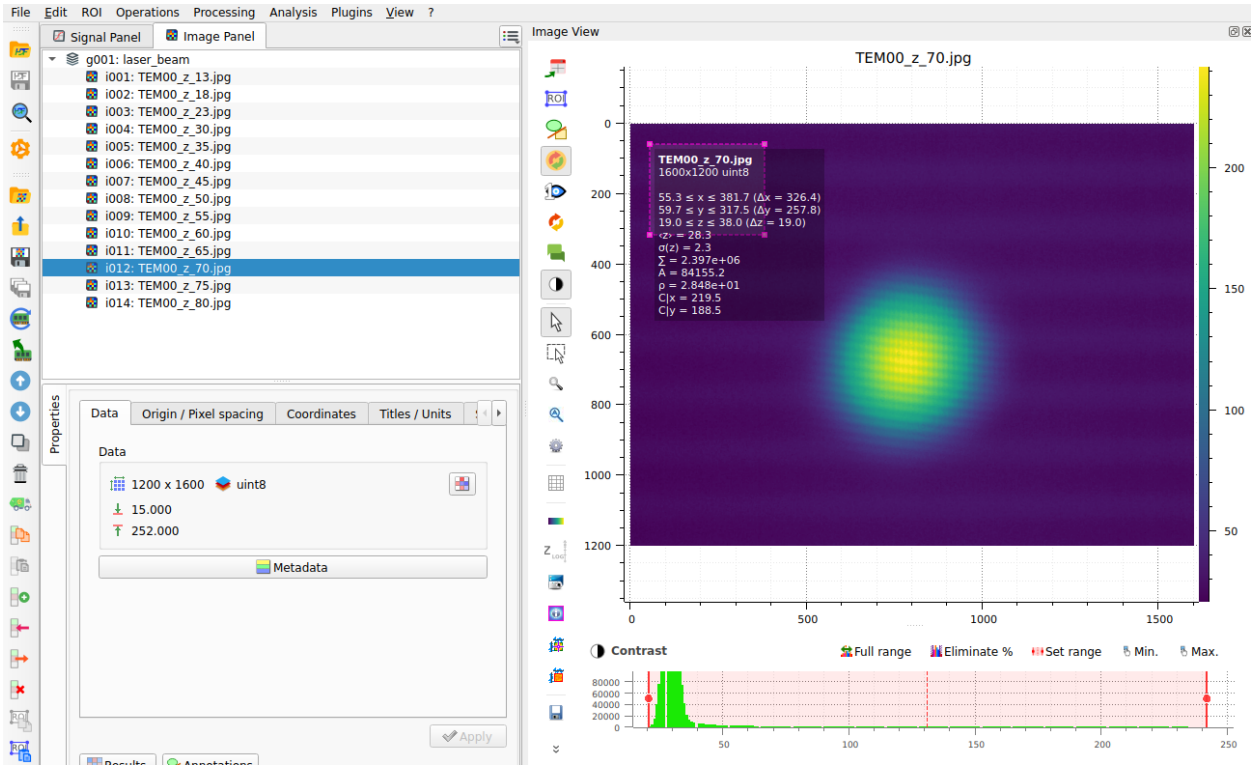


Fig. 82: The “Image statistics” tool in the vertical toolbar.

ping…” menu.

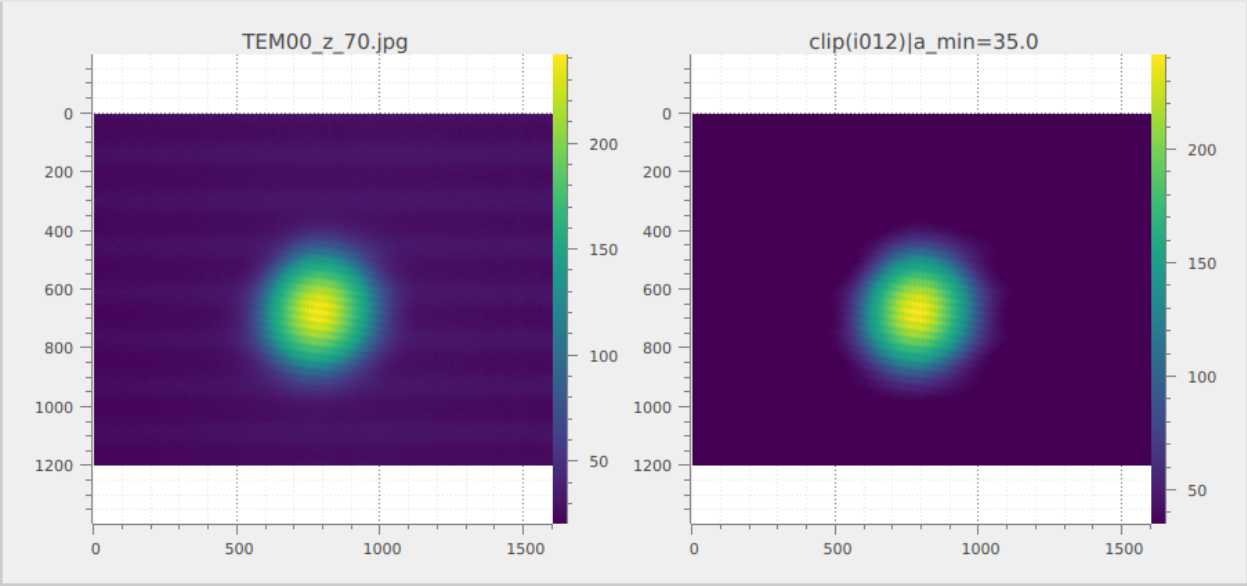


Fig. 83: The two original and the clipped images are displayed side by side in the “Image view” (using the “Distribute on a grid” feature seen previously).

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 Y coordinate of the row to the Y coordinate of the previously computed centroid (i.e., 674) using the “Edit profile parameters” button. See Analyzing Fabry-Perot interferograms: ring detection and circle fitting 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.

> **Note**
>
> If history recording is enabled, the image-processing actions, extracted profile, and Gaussian fit are recorded in the same active history session. See Recording and replaying a processing chain for a cross-panel workflow example.

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 extraction…”. 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.

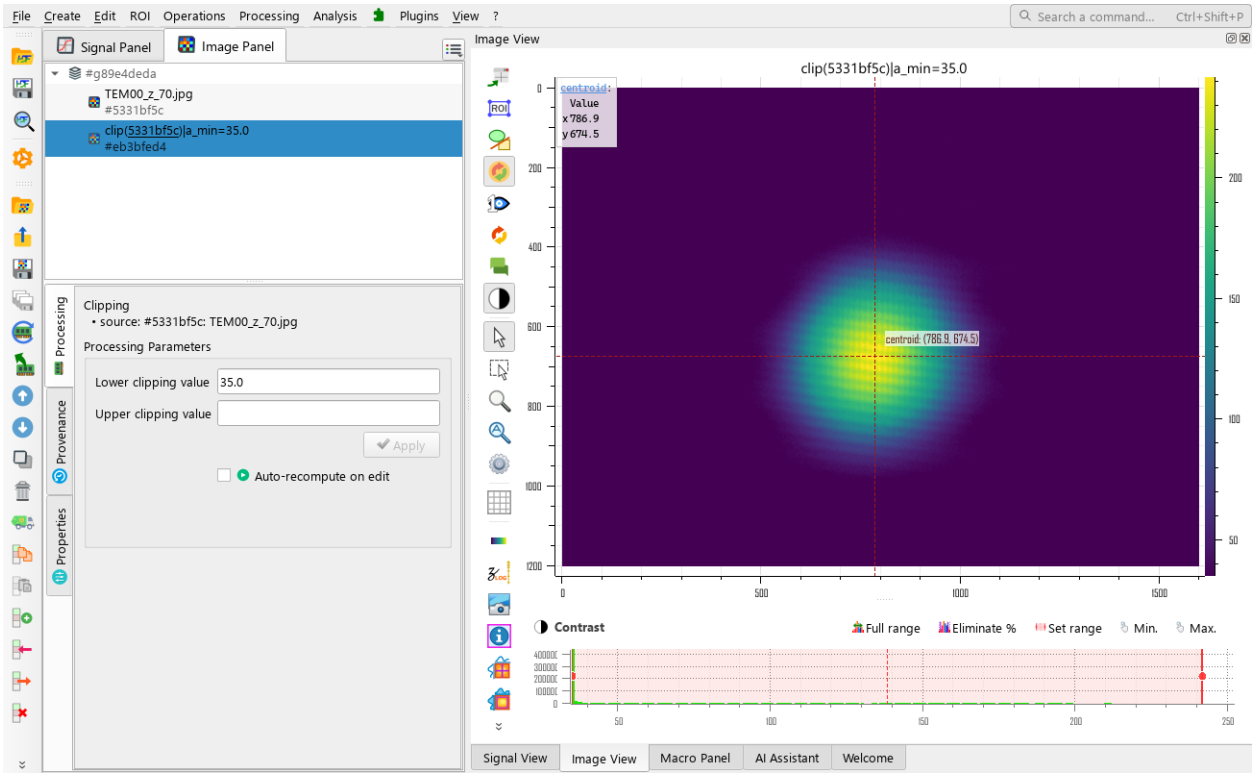


Fig. 84: The centroid position is displayed on the image.

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.

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 (Δx) as a function of the image index (Indices).

We can also calibrate the X and Y axes using "Processing > Axis transformation > Linear calibration…". Here we set

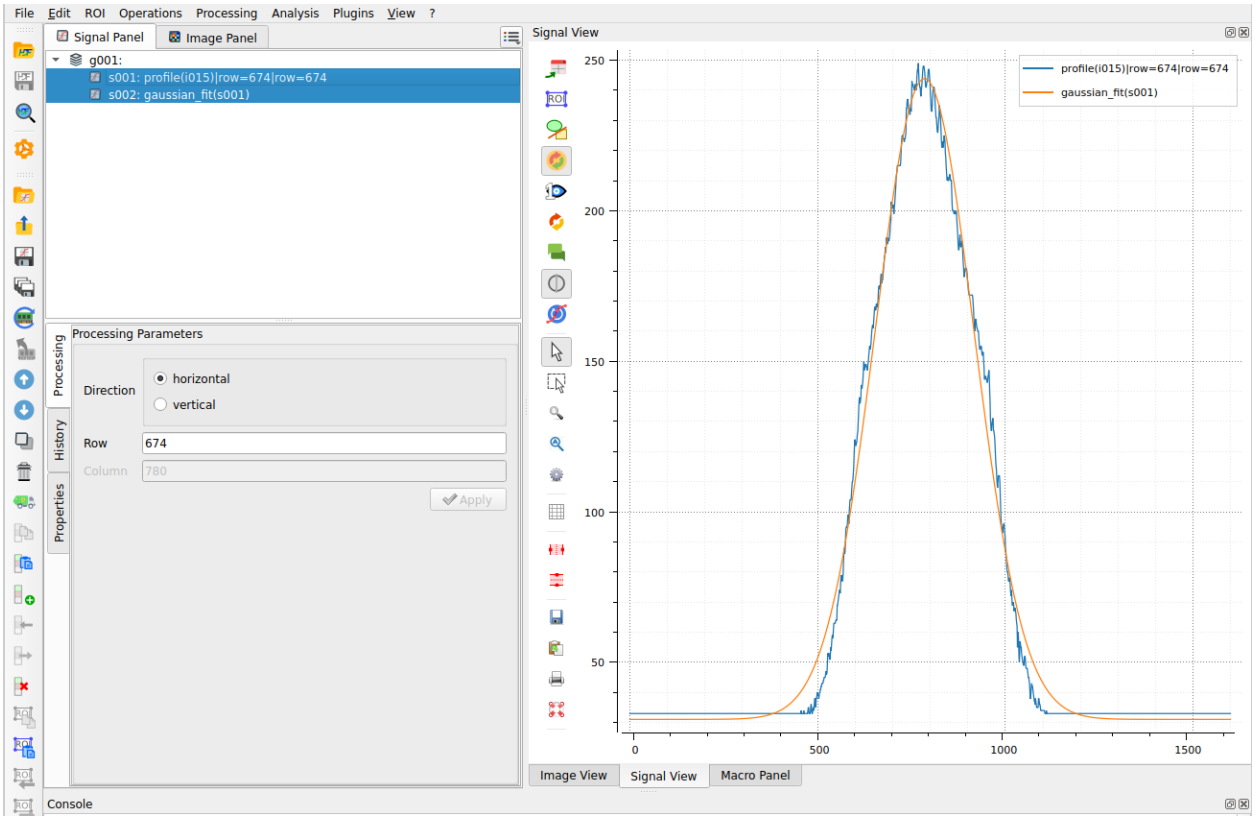


Fig. 85: The intensity profile fitted to a Gaussian function. Here both signals are selected.

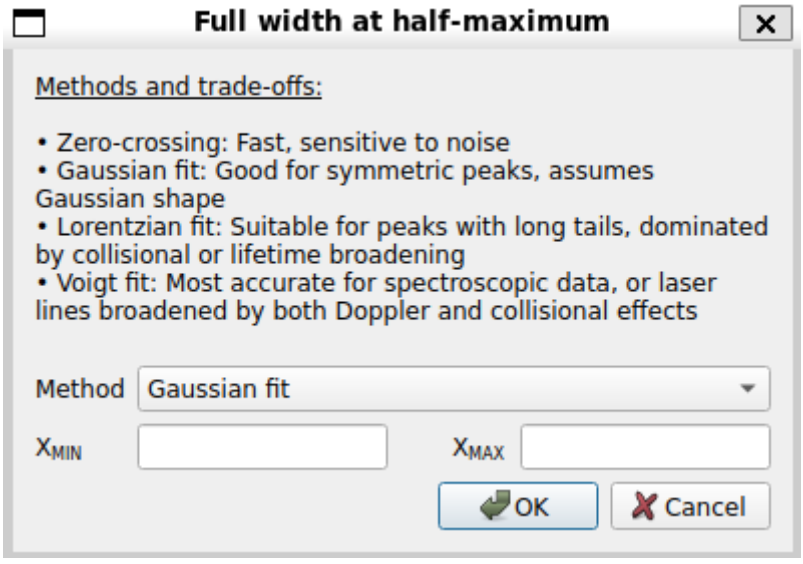


Fig. 86: The popup that allows to choose the method to estimate the FWHM.

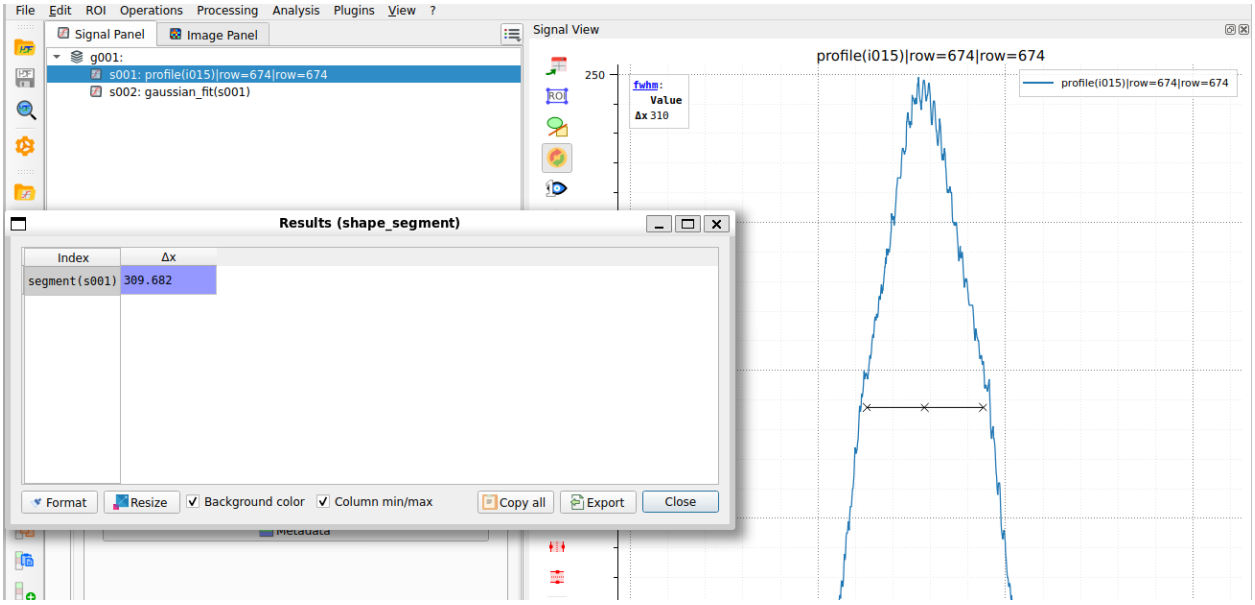


Fig. 87: The FWHM result shown in the popup and over the curve.

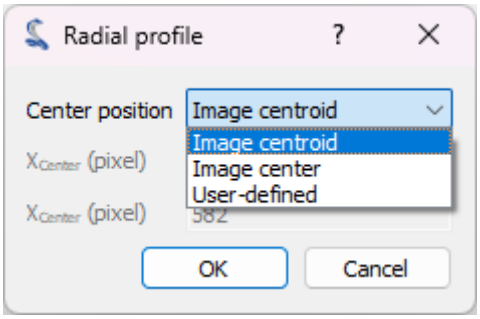


Fig. 88: The options available for the Radial Profile computation.

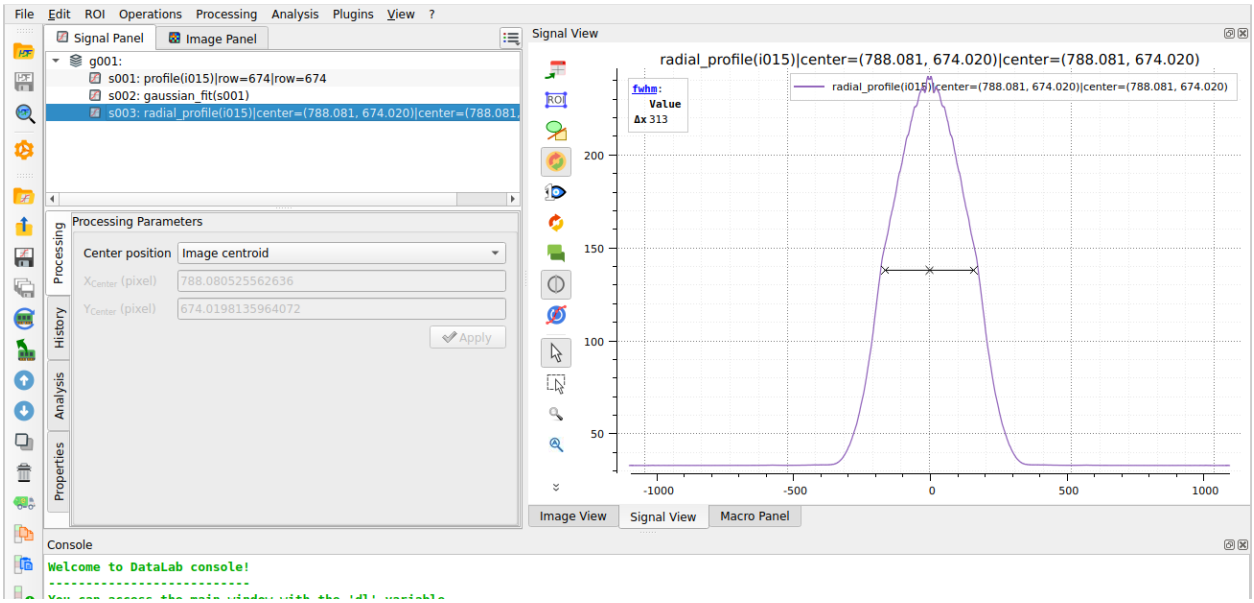


Fig. 89: The radial intensity profile displayed in the "Signal panel". It is smoother than the line profile, because it is computed from a larger number of pixels, thus averaging the noise.

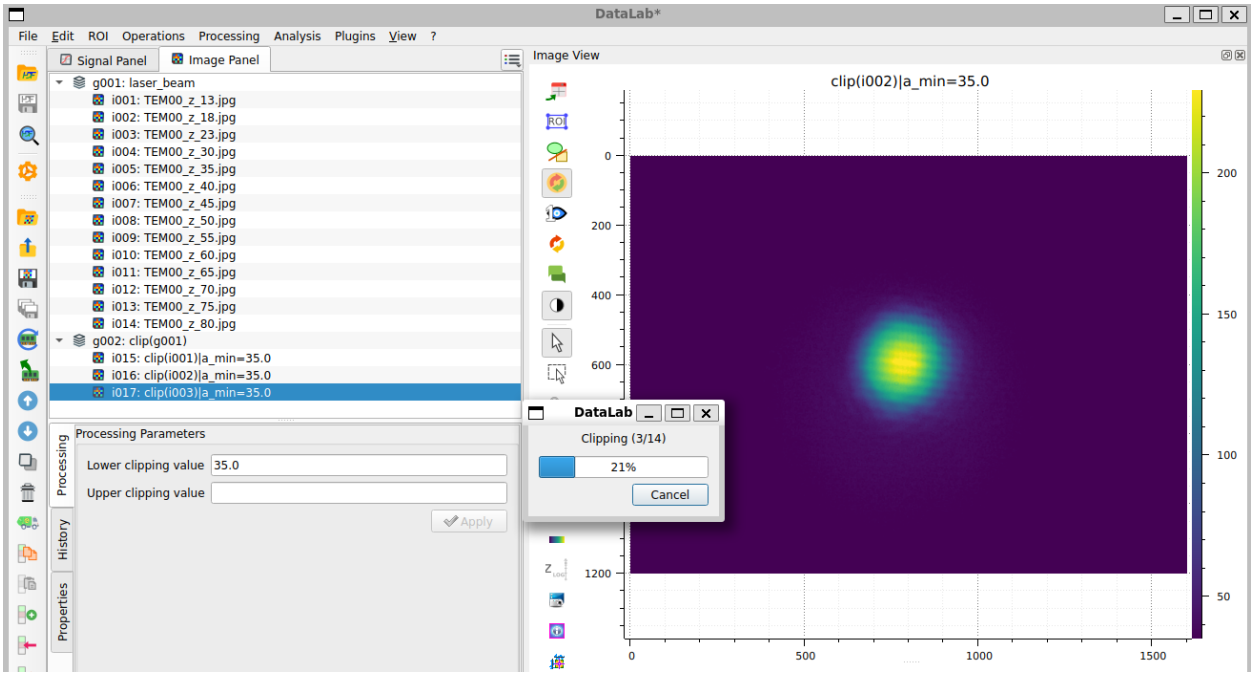


Fig. 90: The clipping applies to all images of the group.

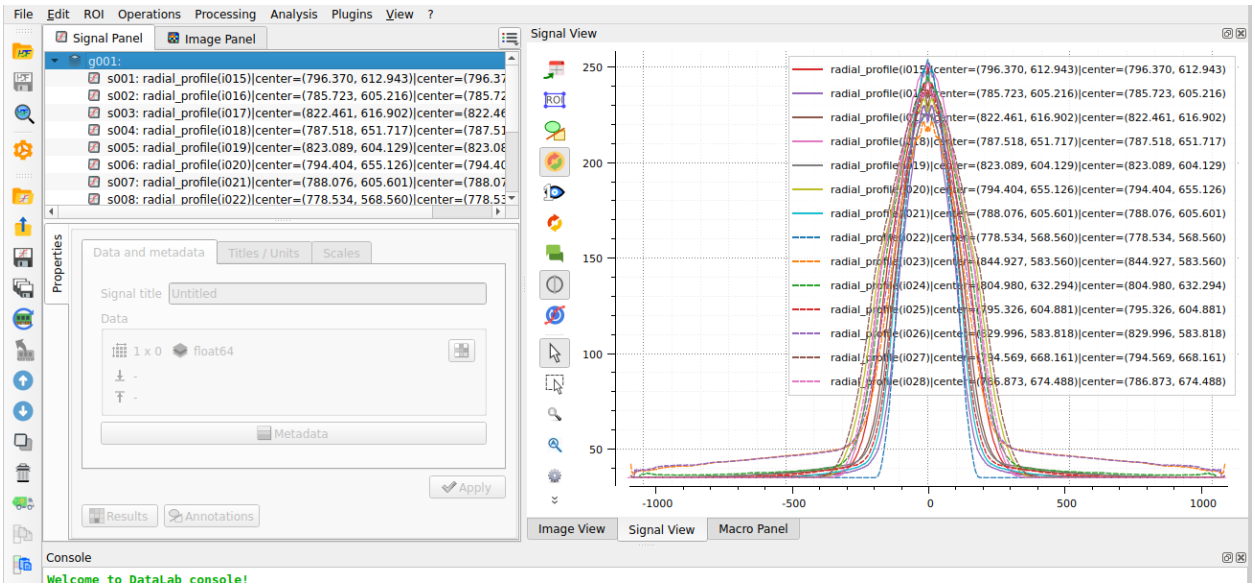


Fig. 91: The “Signal panel” now contains all the radial intensity profiles.

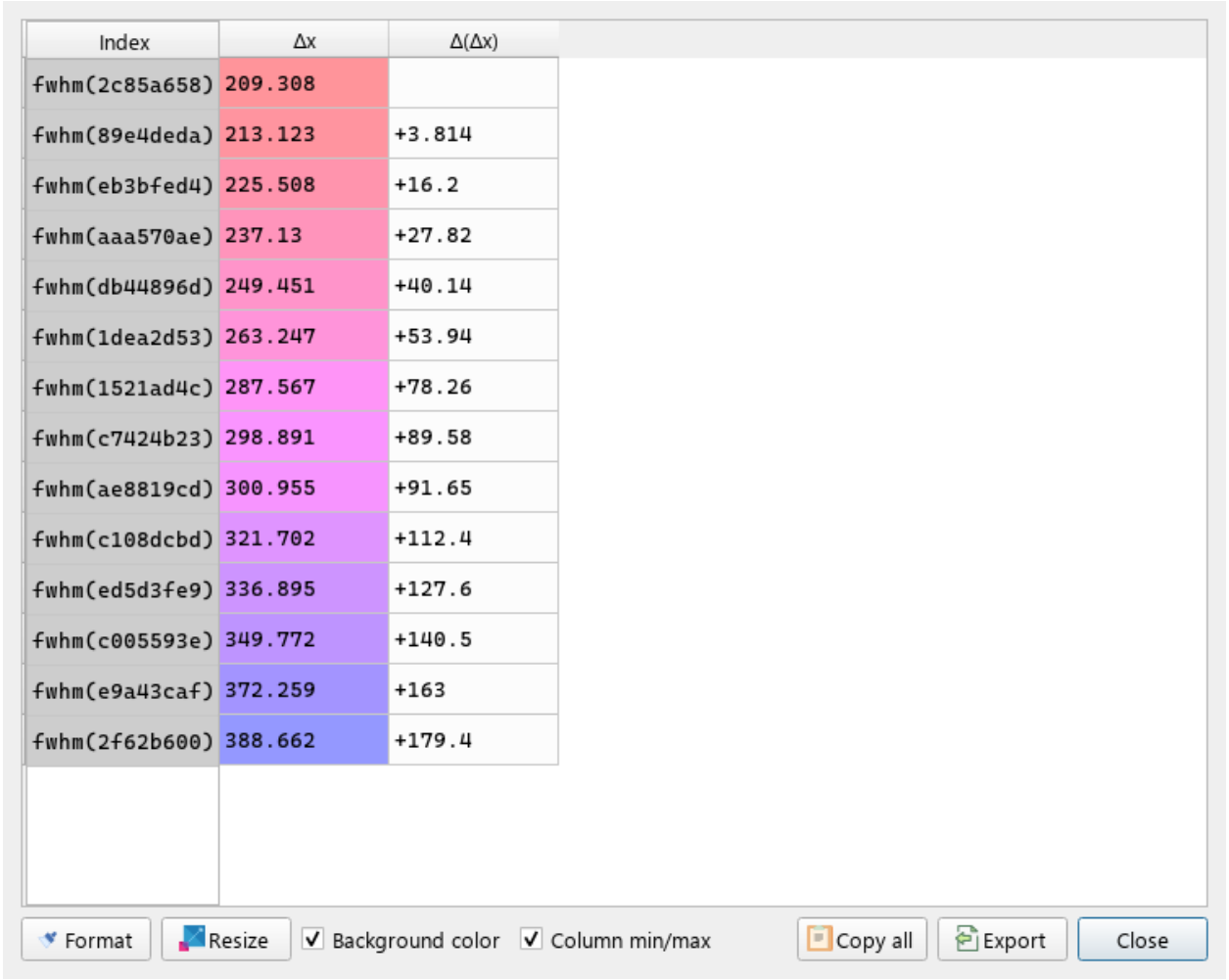


Fig. 92: The “Results” dialog displays the FWHM values for all the profiles.

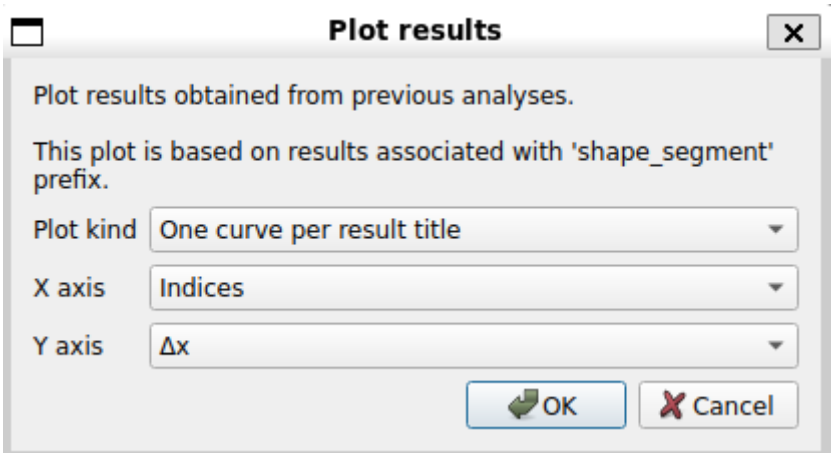


Fig. 93: The "Plot results" feature in the "Analysis" menu.

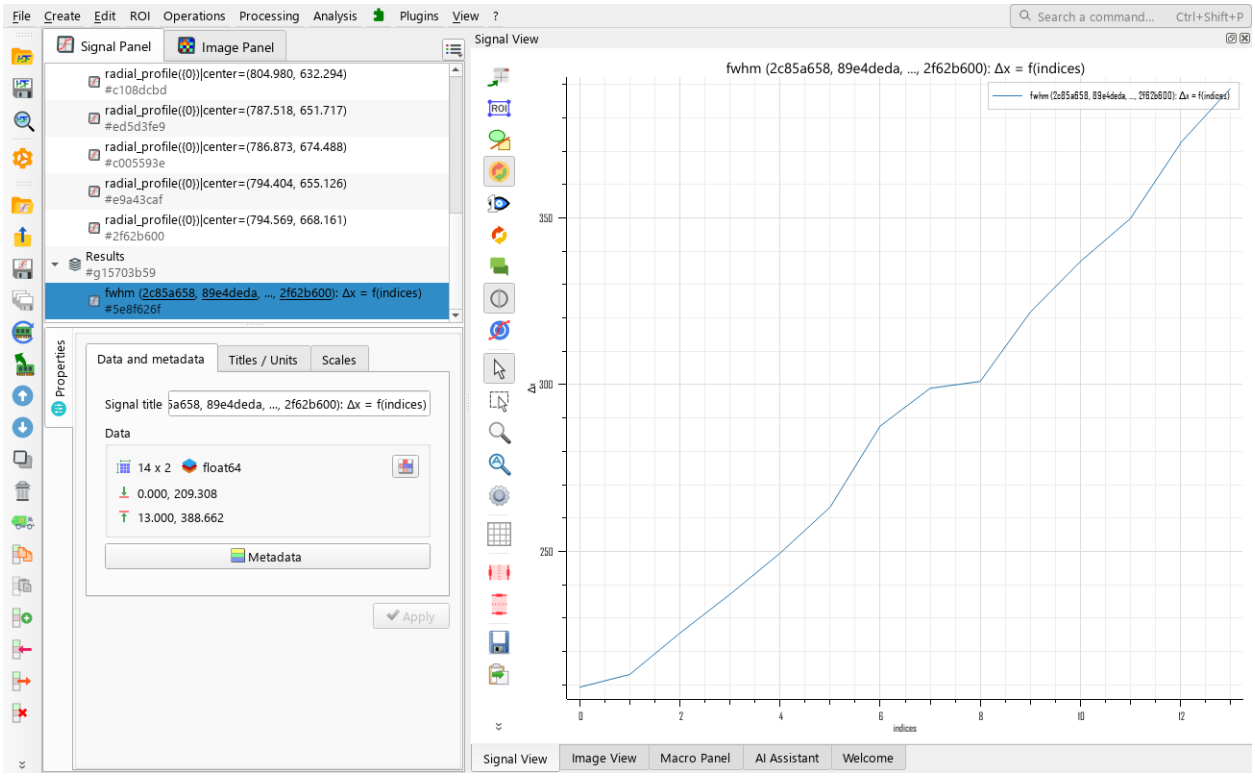


Fig. 94: The beam size as a function of position along the propagation axis (the position is in arbitrary units—the image index).

the X axis to the position in mm (entering the title and unit in the "Properties" group box) using the formula: $X\,[\mathrm{mm}] = 0.5 \cdot i + 10$ where i is the image index.

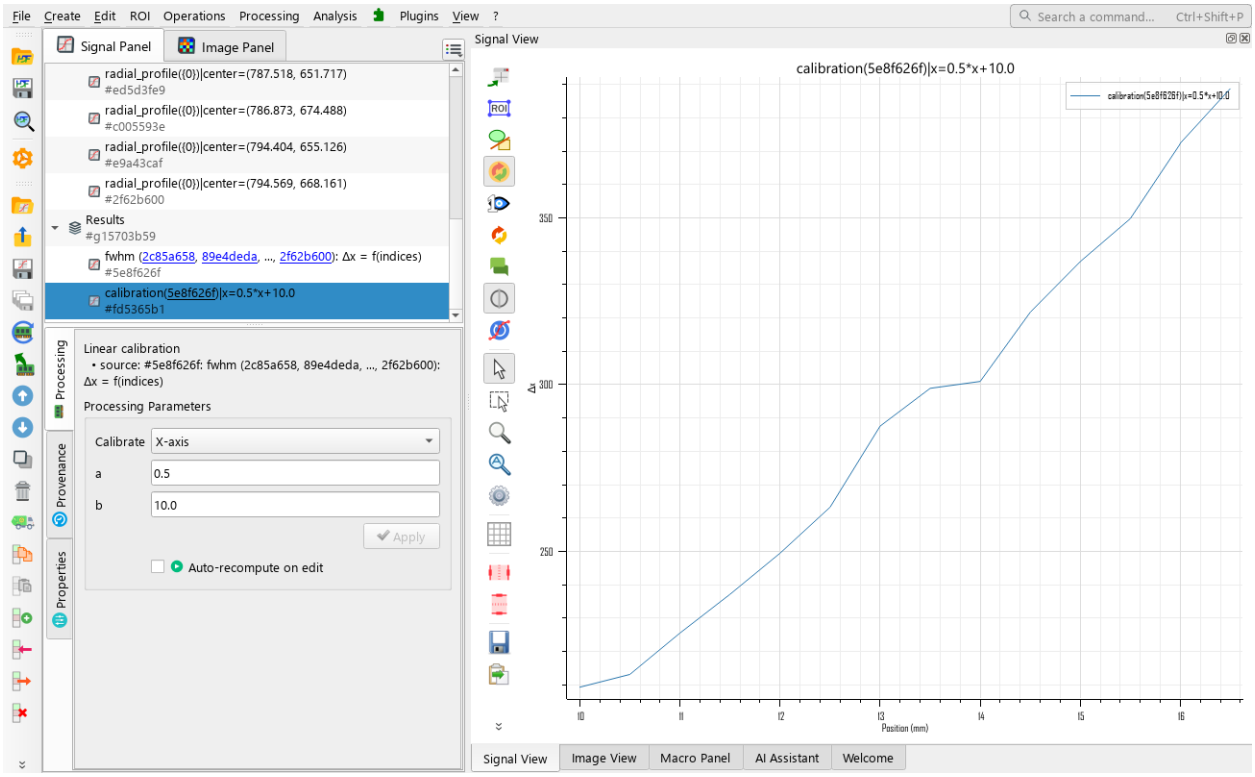


Fig. 95: The calibrated beam size as a function of the position along the propagation axis (the position is now in mm).

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 files…" (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.

Recording and replaying a processing chain

This tutorial shows how to use the History Panel to make a processing workflow reproducible:

- Record a complete processing chain on a test signal
- Inspect the recorded actions and their workspace state
- Replay the chain, and replay it step-by-step with modified parameters
- Duplicate a session to compare two processing variants

> See also
>
> The reference documentation of the panel (toolbar, tree view, persistence rules, chain reconnection) is available in the History Panel section.

Showing the History Panel

The History Panel is visible by default, but could be hidden in your configuration. In this case, to display it, check the "History Panel" entry in the "View" menu.

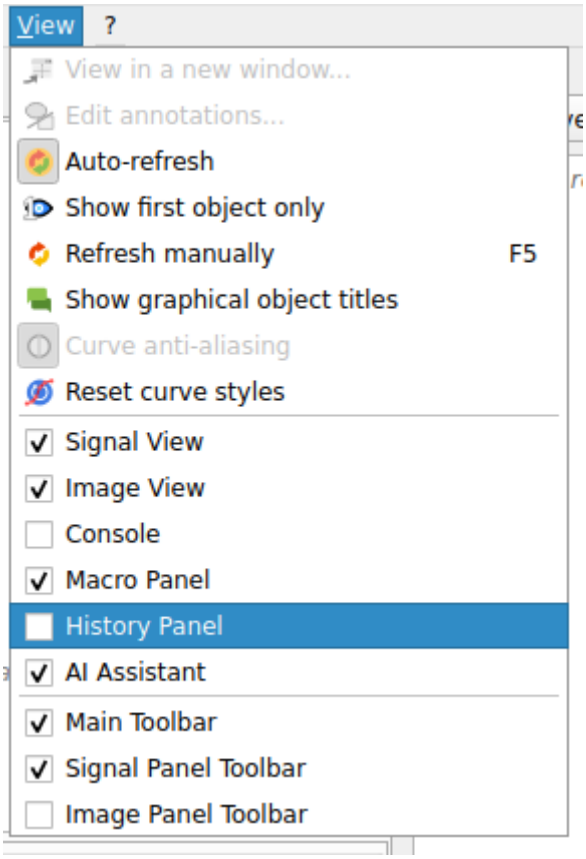


Fig. 96: The "View > History Panel" menu entry, which toggles the visibility of the panel.

The panel is empty when it is first displayed. It is made of three parts: a toolbar at the top, the tree of recorded sessions and actions in the middle (with the "Title", "Date and time" and "Description" columns), and the workspace state tables at the bottom ("Signal" / "Shape" on the left, "Image" / "Dimensions" on the right).

Like the other panels of DataLab, the History Panel is dockable: it can be moved to any side of the main window, or detached as a floating window. Here, it is docked on the left side, next to the Signal Panel, so that the recorded actions remain visible while working on the data. We use this configuration in the rest of the tutorial, but you can choose any layout that suits your workflow.

Recording a processing chain

Recording is disabled by default: the History Panel only starts collecting actions once the Record mode button is enabled in its toolbar.

The first object created after enabling the recording opens a new session, which acts as a container for the whole processing chain. From then on, every action producing or modifying an object – creating a signal, running a computation, removing an object – is appended to that session.

When a new object is created afterwards, DataLab asks whether the current session should be closed and a new one started. Since we want to record a single, continuous processing chain, we answer No to keep appending to the session

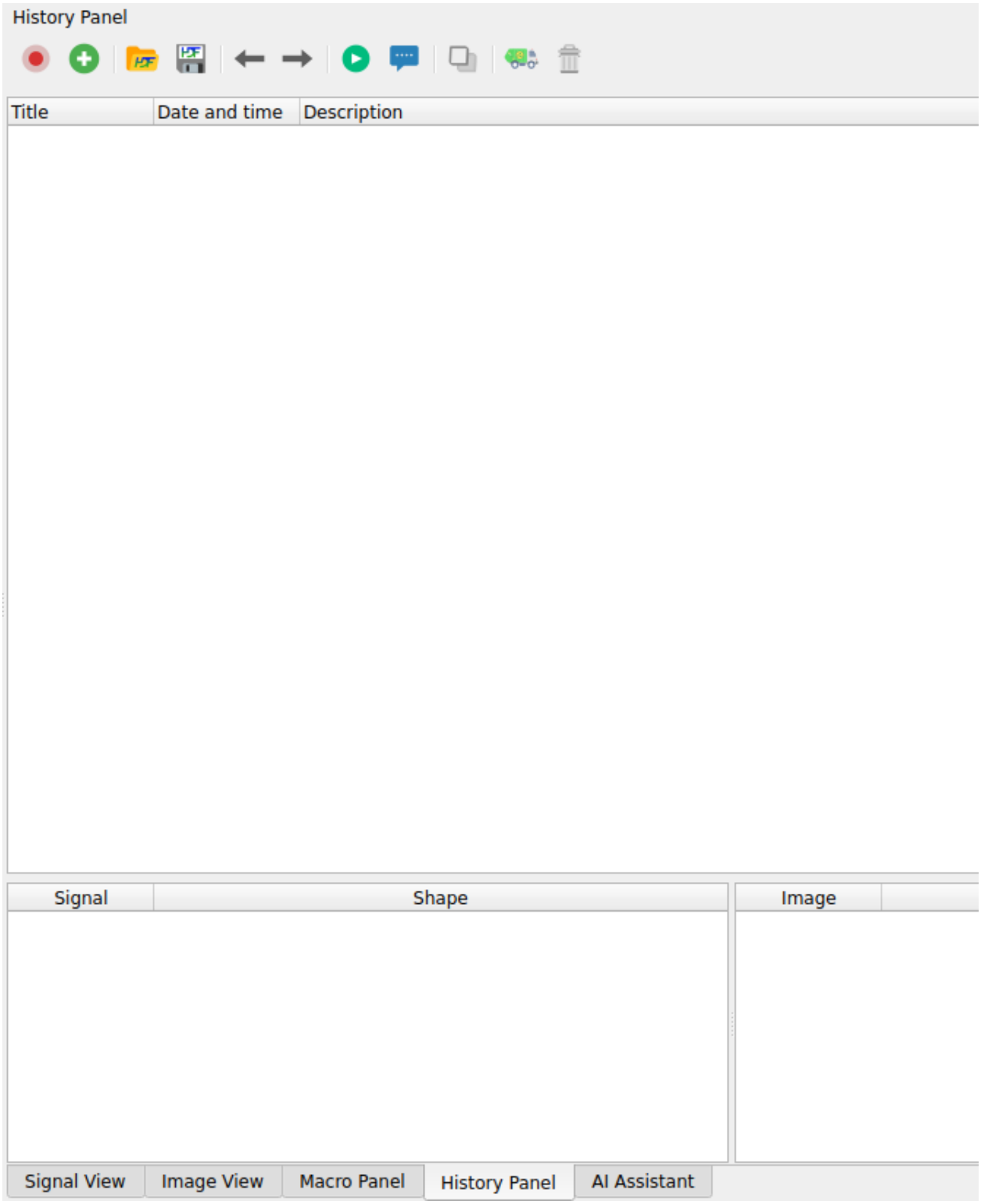


Fig. 97: The empty History Panel: toolbar, action tree and workspace state tables.

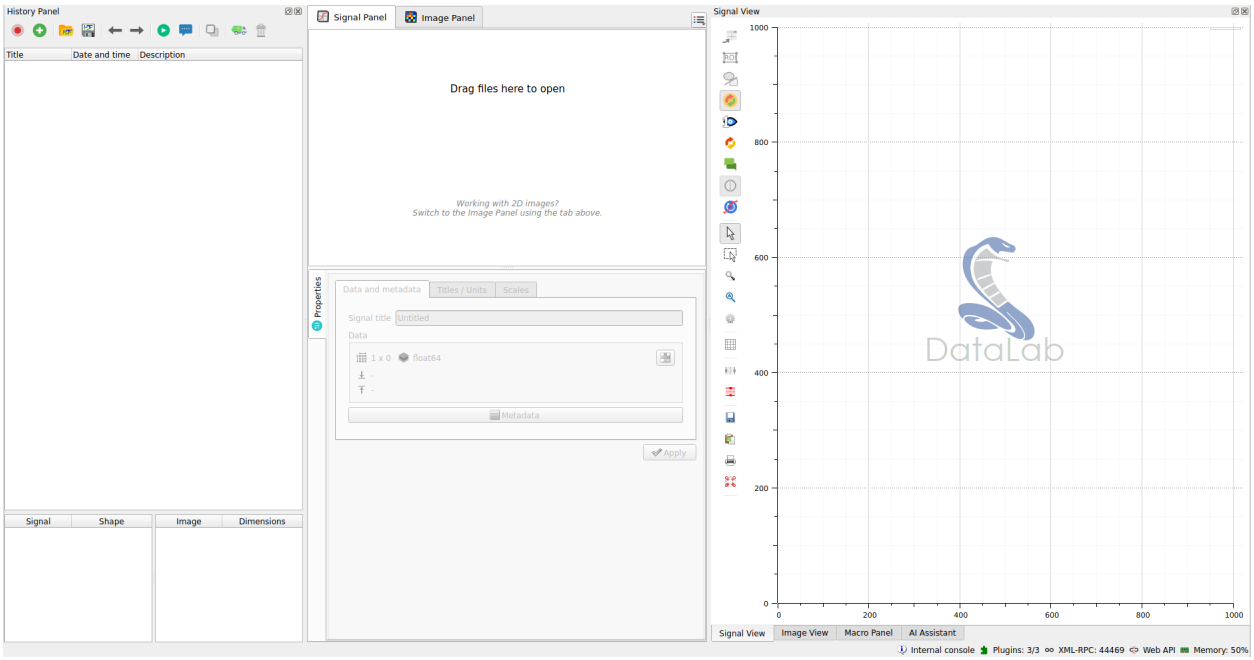


Fig. 98: The History Panel docked on the left side of the main window.

already in progress.

To illustrate the recording of a processing chain, let's create a test using the "Create > Gaussian" menu entry and then add to it some noise using "Processing > Noise addition > Add Gaussian noise…".

The tree now contains a Processing session with the two recorded actions: the creation of the Gaussian signal and the addition of Gaussian noise. The selected noise action exposes its parameters in the description column. The Signal Panel shows both the original and resulting signals, while the Processing Parameters panel displays the seed, mean and standard deviation used for the selected action.

To create a longer chain, we can apply a few more processing steps, for example a fitting. Once done, the action is recorded in the History Panel, and the resulting signal is displayed in the Signal Panel.

Familiarizing with the history panel commands

Now we have a very simple processing chain, which is enough to illustrate the main features of the History Panel. The toolbar provides the following commands:

- Record mode toggles the recording of subsequent actions.
- New session starts a separate history session.
- Open history file and Save history file load or save a standalone `.dlhist` history file.
- Previous step and Next step select the adjacent action in the current session.
- Replay applies the selected session or action without showing parameter dialogs, whereas Step-by-step replays it while allowing parameters to be edited at each step.
- Show execution provenance shows the recorded provenance of the last successful execution of the selected computation action (see Provenance of results).
- Duplicate copies the selected session, or the complete session containing the selected action, to compare a processing variant.

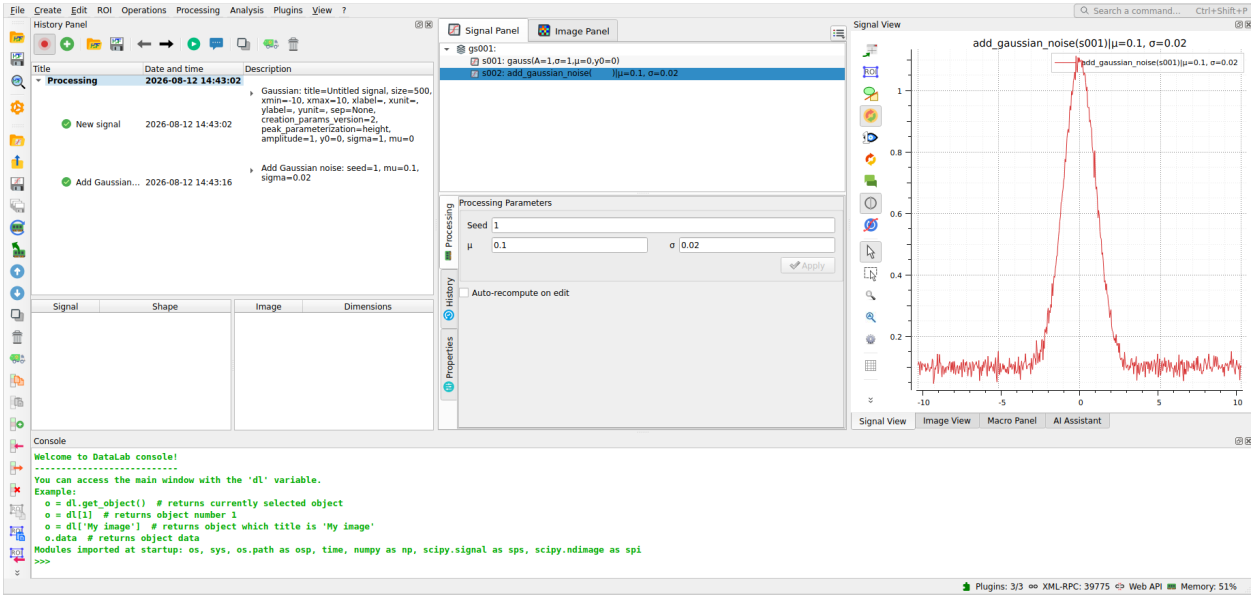


Fig. 99: The History Panel after creating a Gaussian signal and adding noise to it.

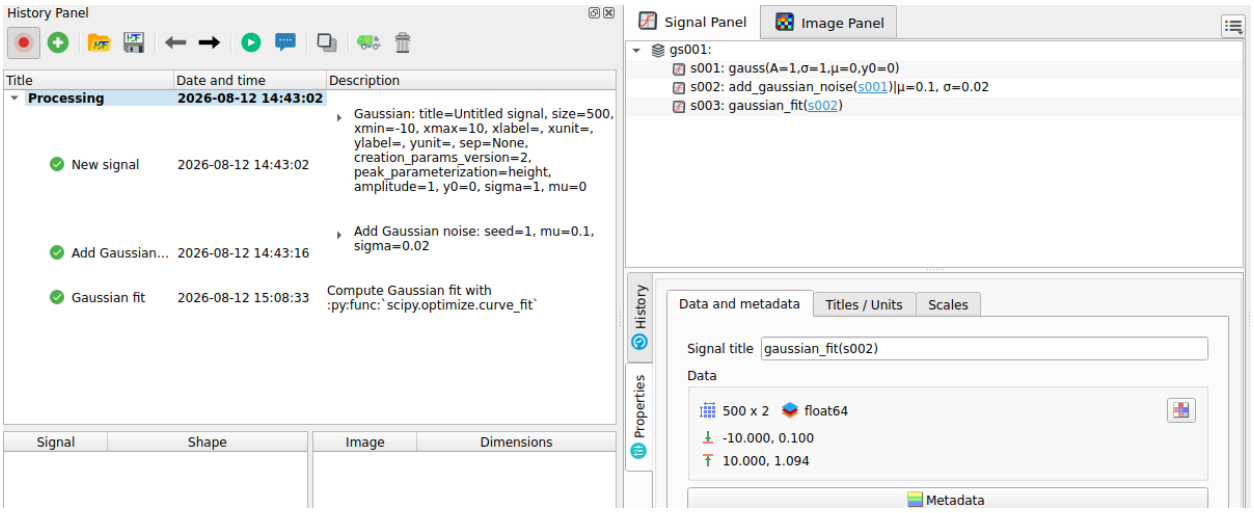


Fig. 100: The History Panel after fitting the noisy signal.

- Remove incompatible removes actions that cannot be directly applied in the current workspace (i.e. because a signal has been deleted), and Delete removes the selected history entry.

When an action is selected in the tree, the corresponding resulting signal is displayed in the Signal Panel.

Now select the fitting signal in the Signal Panel, and delete it using the "Delete" button in the toolbar or pressing the "Delete" key. The fitting action is still available in the History Panel, as it can be replayed using the Replay button. In addition, the delete action has been recorded in the history, but it is grayed: the reason is that the fitting signal has been deleted, so the action cannot be directly applied in the current workspace.

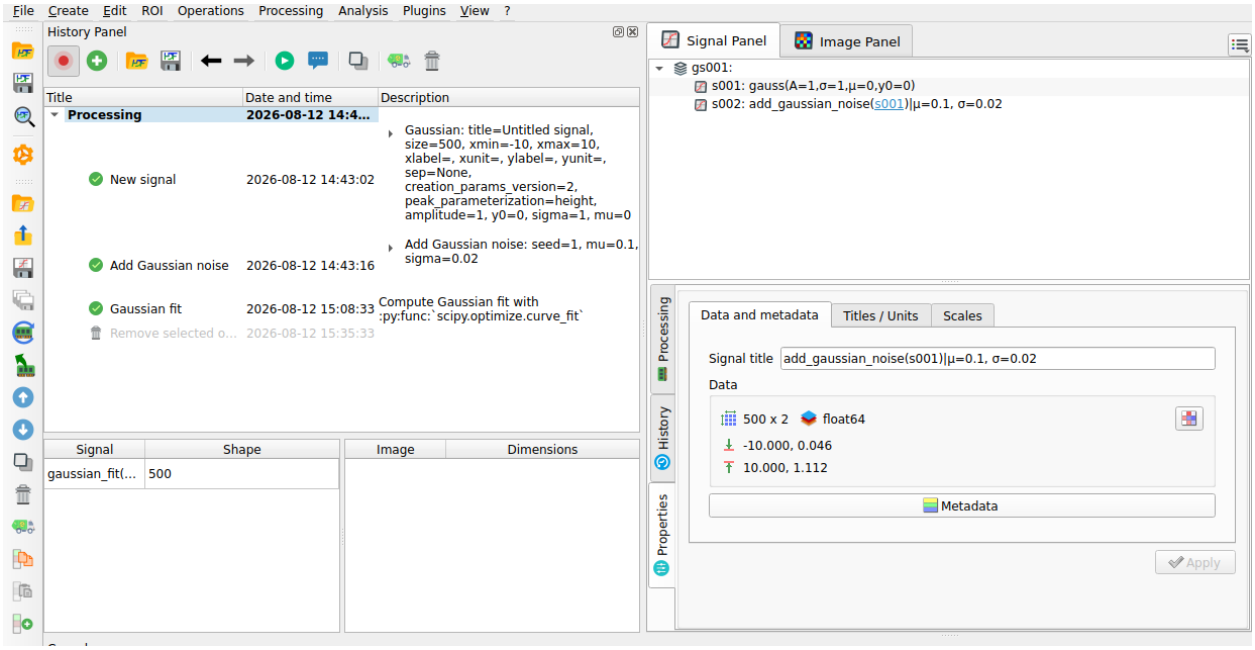


Fig. 101: The deleted fitted signal is no longer listed in the Signal Panel. The recorded deletion is grayed in the History Panel because it cannot be replayed in the current workspace.

We can now replay the fitting action: the resulting signal is recreated in the Signal Panel and the "delete" action is no longer grayed, as it can now be applied to the newly created signal.

Duplicating the chain to compare variants

We can now duplicate the chain to compare two processing variants. First of all, select the "Remove selected objects" entry in the History Panel and press the Delete key. Then select the Processing session and click Duplicate. DataLab creates a new Processing Copy session and a `Copy - Processing` group in the Signal Panel. This group contains independent copies of the signals used and produced by the original chain.

To create a variant, select the Processing Copy session and click Step-by-step. DataLab displays a parameter dialog for each action. In the Add Gaussian noise dialog, change the noise level, for example set `sigma` to `0.05`, then accept the remaining dialogs. The copied signal chain is recomputed in place: its noisy signal and fit now use the new noise level, while the original Processing session remains unchanged.

This can be useful to compare the effect of the noise level on the fitting result, or to explore other processing variants. In this example, selecting both the original and copied fitting signals in the Signal Panel shows a difference between them, but it is quite slight.

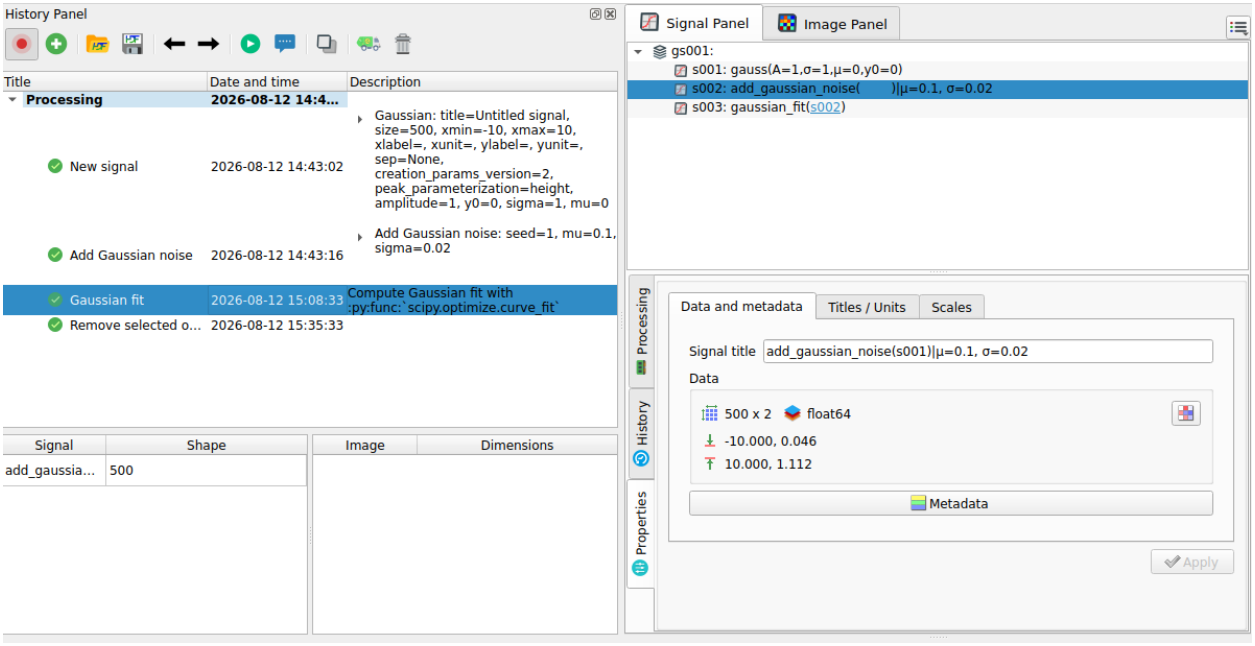


Fig. 102: Replaying the fit recreates the fitting signal. All recorded actions are again compatible with the current workspace.

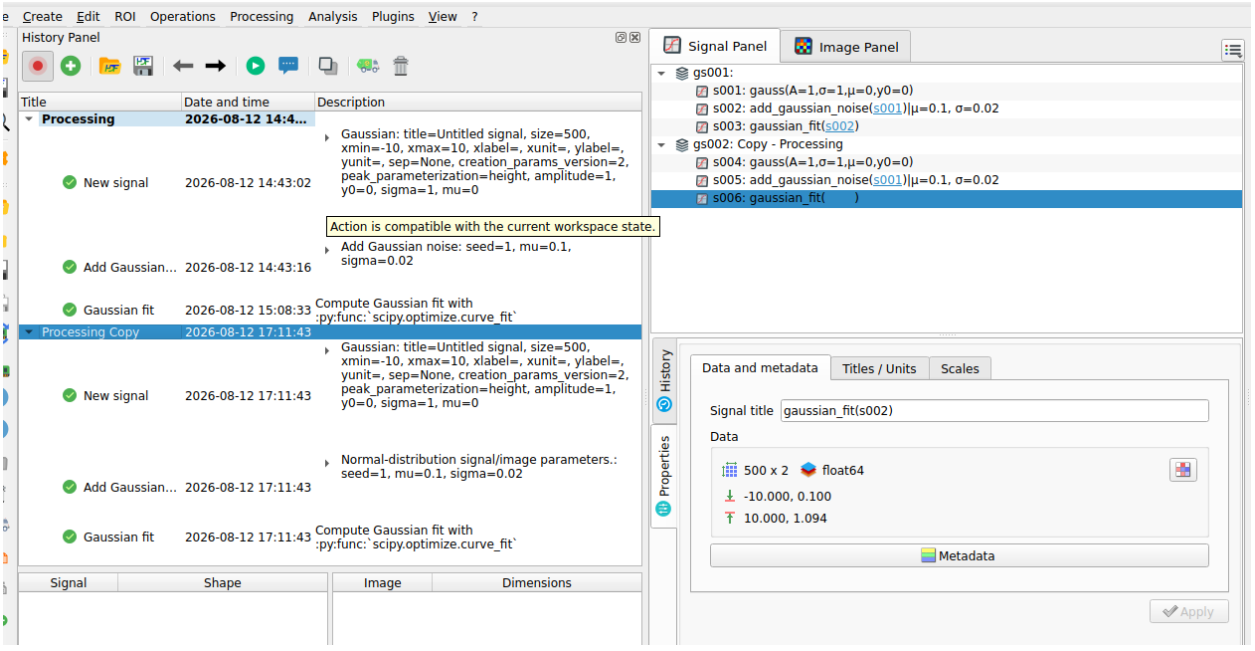


Fig. 103: The Processing Copy session contains the duplicated actions. The `Copy - Processing` group contains the independent signal copies.

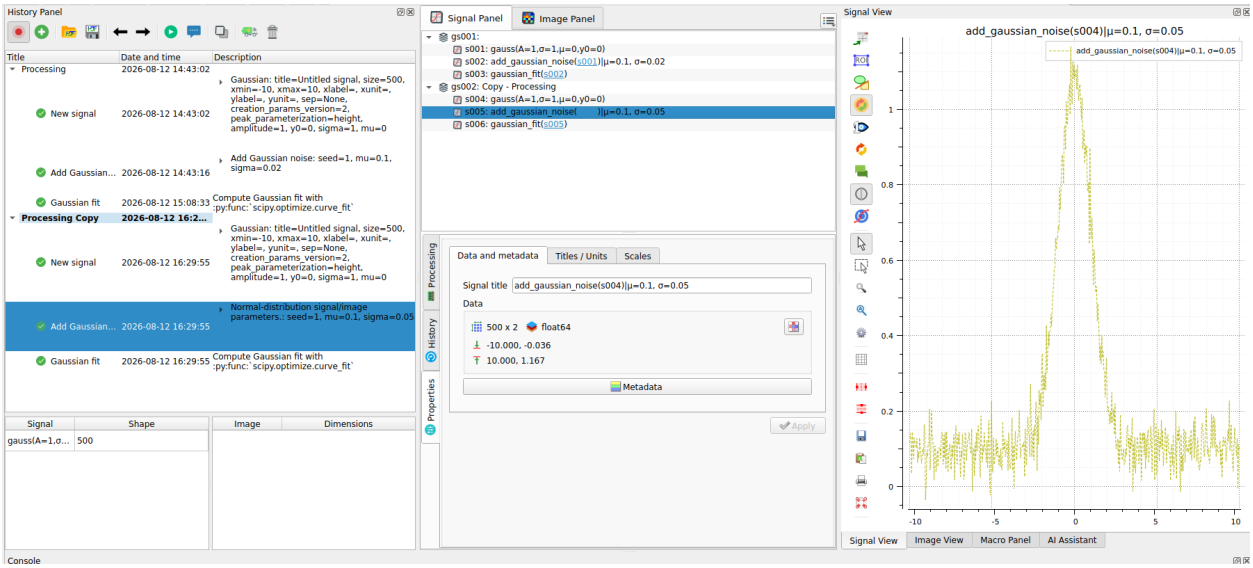


Fig. 104: The selected noise action in Processing Copy uses `sigma = 0.05`. The original chain and the modified copy remain available side by side.

Recording an image-to-signal workflow

The History Panel records workflows across both data panels in the same session. Unlike the Signal Panel and Image Panel, it is not divided by data type: its chronological tree shows image operations and any signals that they produce together.

To illustrate this behavior, we will start from a clean workspace. Alternatively, click New session in the History Panel to record it in a new session of your current workspace.

In both cases, activate the history recording and switch to the Image Panel.

Create or open a test image and apply a first noise-reduction operation, for example Processing > Noise reduction > Moving average…. As you can see, for images the history is recorded in the same way that it is for signals.

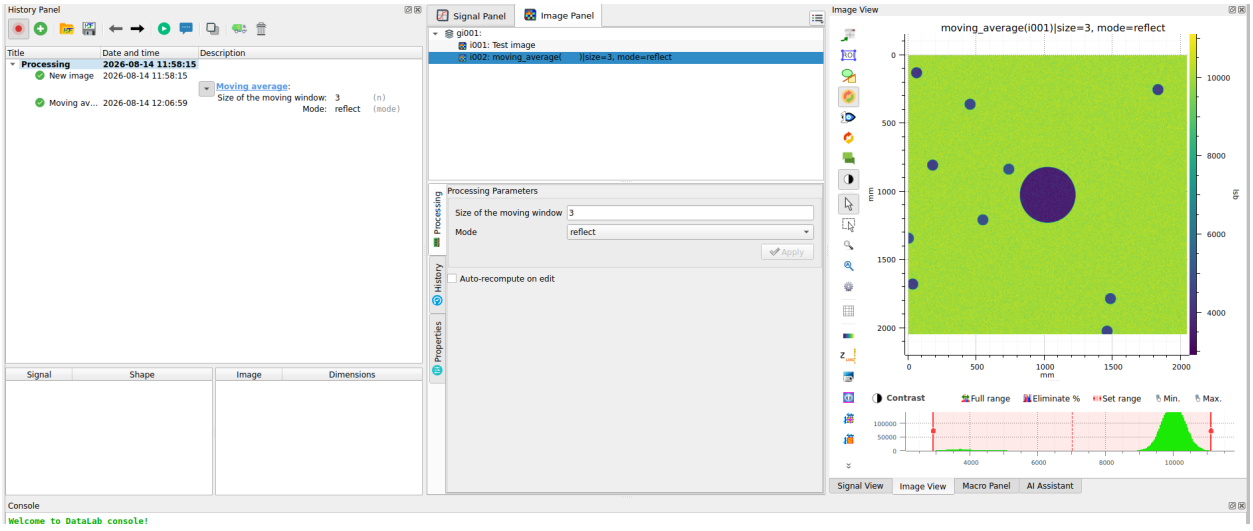


Fig. 105: The new image and its moving-average result are recorded in the same History Panel session.

Select the filtered image and choose Analysis > Intensity profiles > Line profile…. Set the horizontal line in the profile dialog, then confirm it. DataLab adds the extracted profile to the Signal Panel, while the History Panel appends the Line profile action to the same session after the image-processing actions.

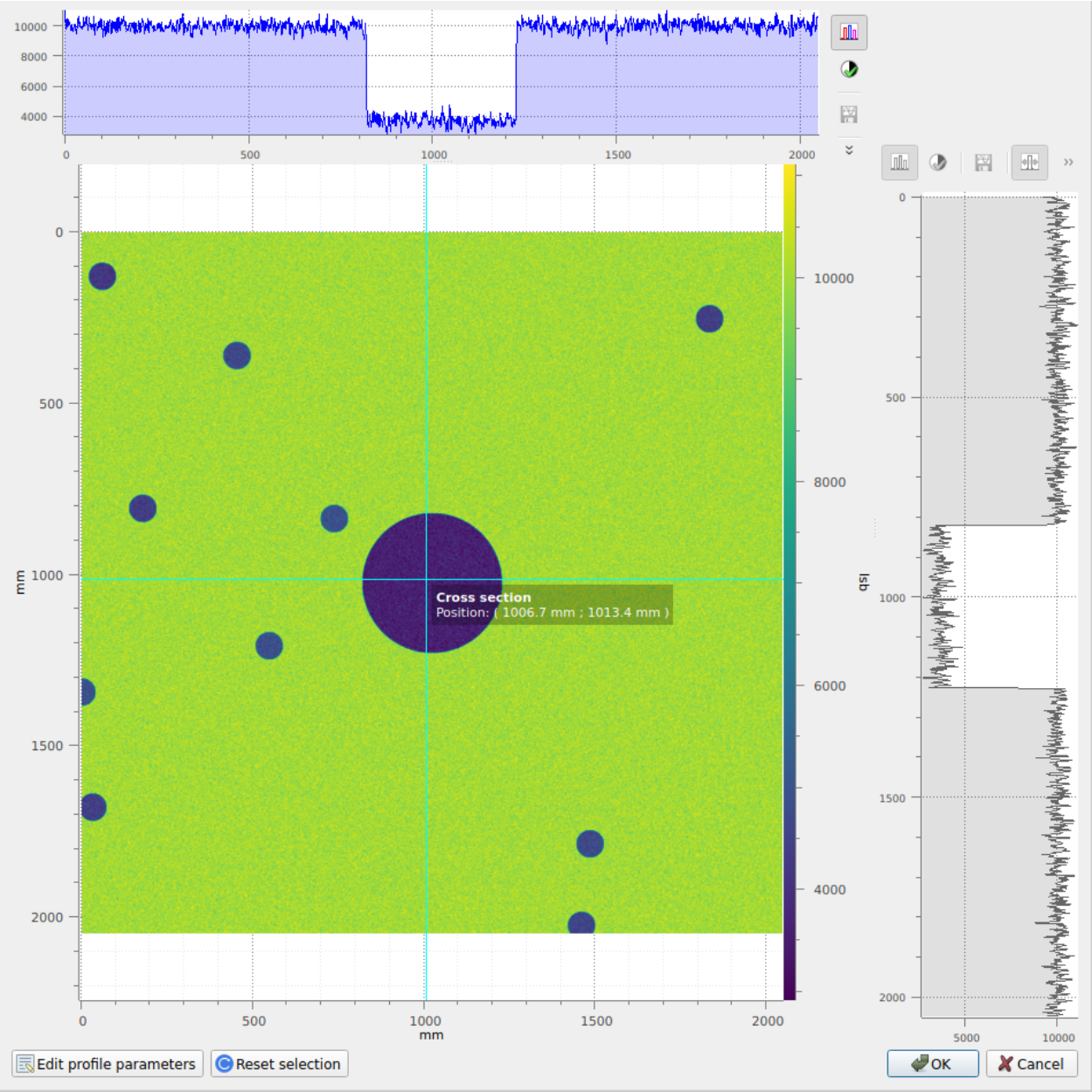


Fig. 106: Choose the horizontal line used to extract the profile from the filtered image.

This single history session therefore captures the full workflow: image creation, noise reduction, and conversion of image data into a signal. Selecting the image actions in the History Panel activates the Image Panel; selecting the profile action activates the Signal Panel and selects the resulting signal.

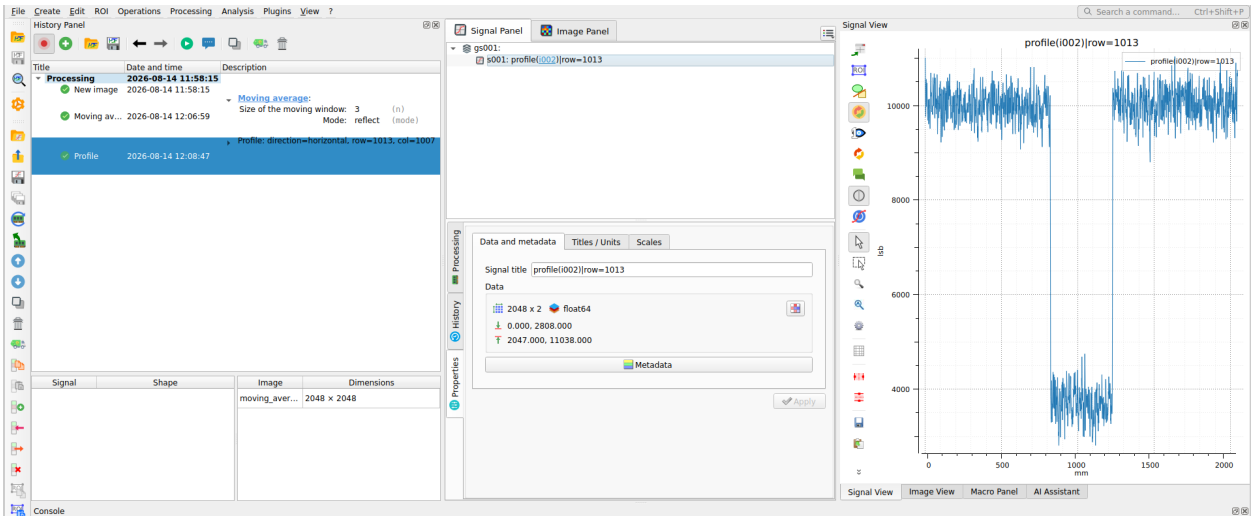


Fig. 107: The History Panel records the image operations and the resulting signal in one chronological session.

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]
```

(continues on next page)

(continued from previous page)

```
        differences = np.abs(values - central_pixel)
        mean_difference = np.mean(differences)
        if mean_difference == 0:
            return central_pixel
        weights = np.exp(-differences / mean_difference)
        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 Automated detection of defects and particles on images 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 > Macro Panel" menu.

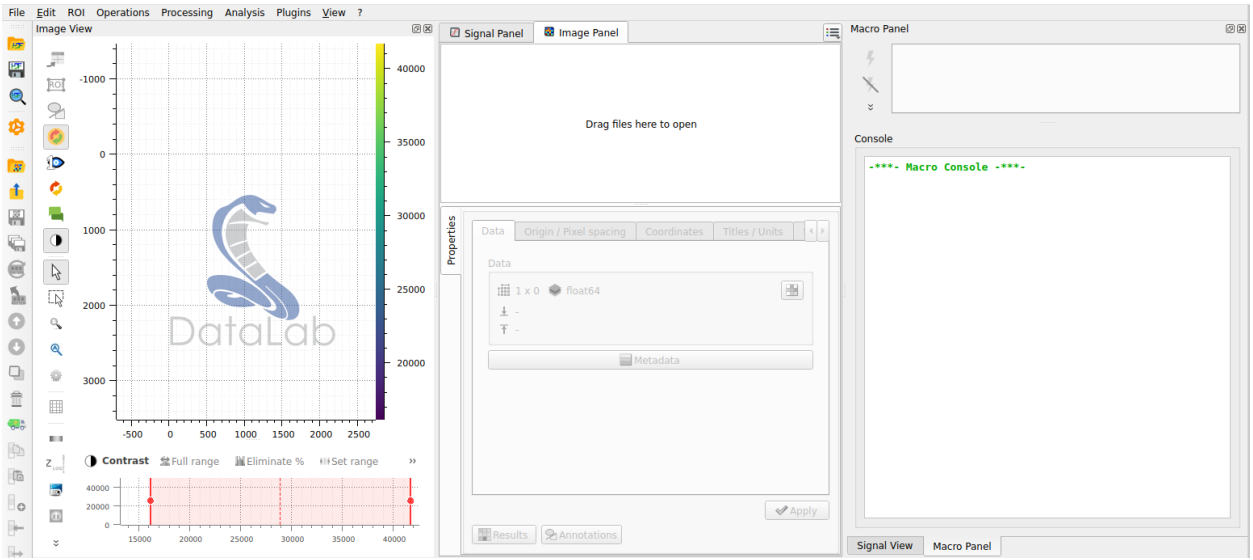


Fig. 108: The window layout of DataLab reorganized to have the "Image Panel" on the left and the "Macro Panel" on the right.

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).

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:

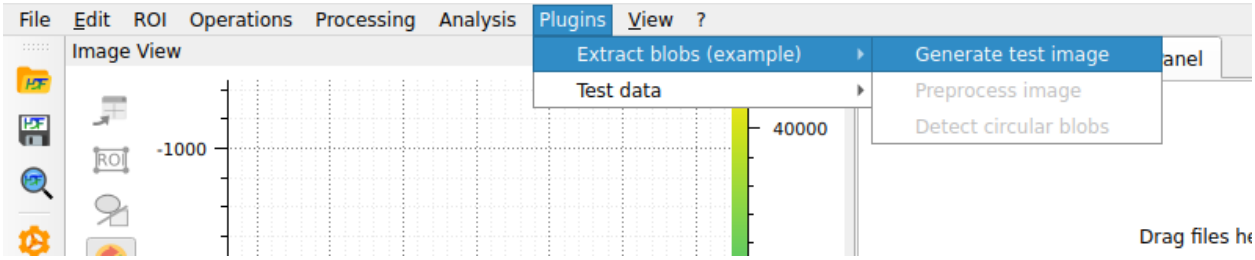


Fig. 109: We generate a new image using the “Plugins > Extract blobs (example) > Generate test image” menu.

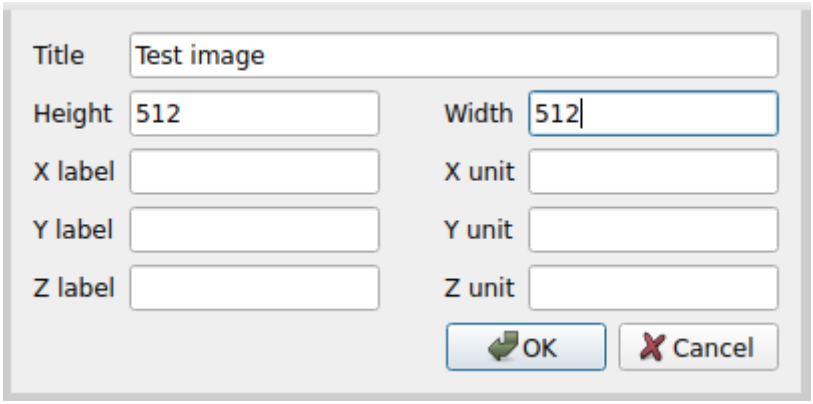


Fig. 110: We select the size for the image and click on “OK”.

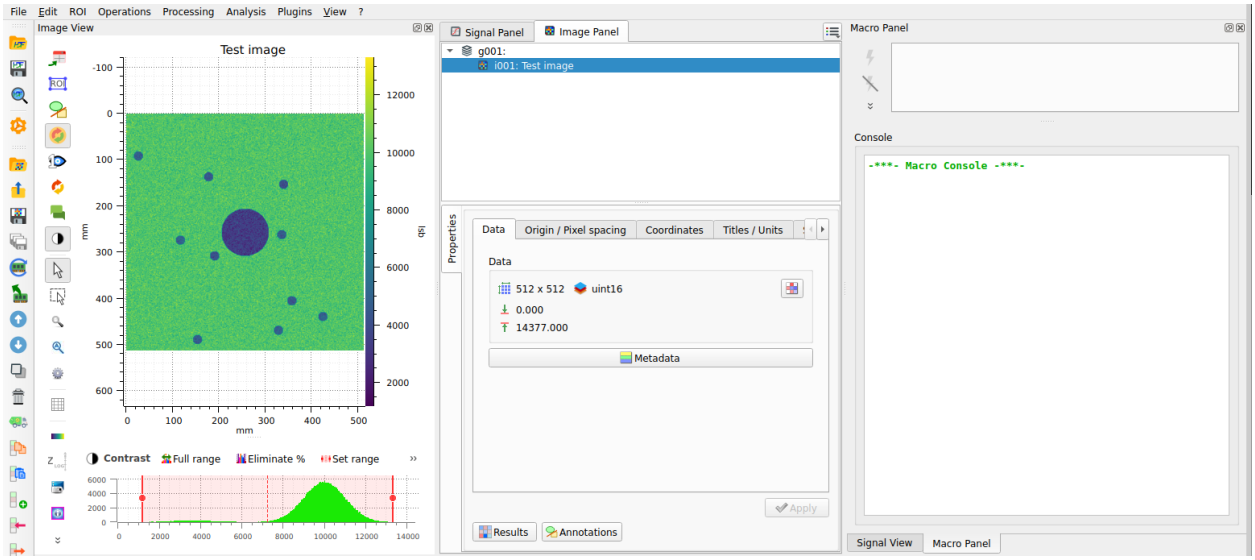


Fig. 111: The generated image in the “Image Panel”.

- 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 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
```

(continues on next page)

(continued from previous page)

```
# 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)
    mean_difference = np.mean(differences)
    if mean_difference == 0:
        return central_pixel
    weights = np.exp(-differences / mean_difference)
    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 'macro_01' macro...]`: the macro-command starts
  - `Connecting to DataLab XML-RPC server...OK [...]`: the proxy is connected to DataLab
  - `---[...]---[# <== 'macro_01' macro has finished]`: the macro-command ends

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

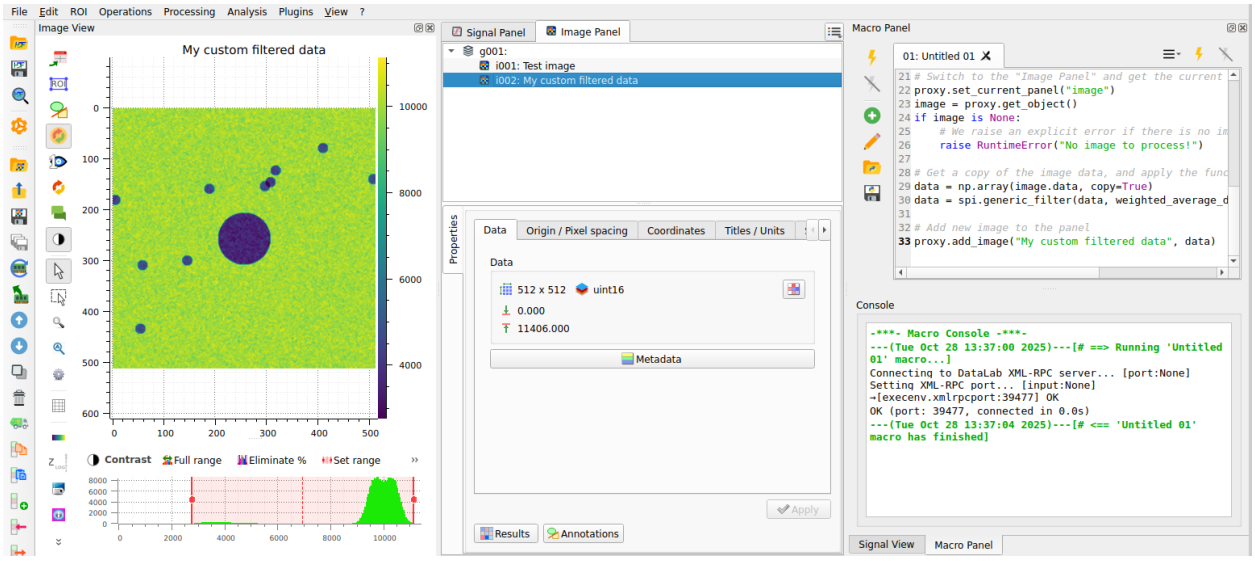


Fig. 112: When the macro-command has finished, we can see the new image in the "Image Panel". Our filter has been applied to the image, and we can see that the noise has been reduced.

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:

```
# -*- 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
```

(continues on next page)

(continued from previous page)

```python
from sigima.client import SimpleRemoteProxy

# %% Connecting to DataLab current session

proxy = SimpleRemoteProxy()

# %% 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.
    """

    def filter_func(values: np.ndarray) -> float:
        """Filter function"""
        central_pixel = values[len(values) // 2]
        differences = np.abs(values - central_pixel)
        mean_difference = np.mean(differences)
        if mean_difference == 0:
            return central_pixel
        weights = np.exp(-differences / mean_difference)
        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”.

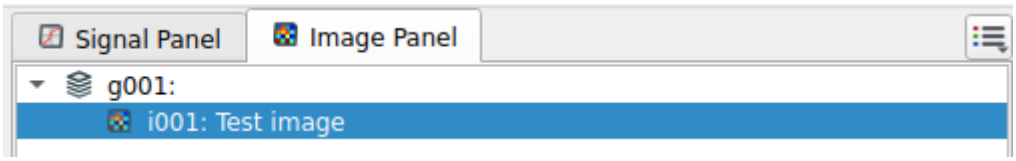


Fig. 113: The first image in the “Image Panel” selected.

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.

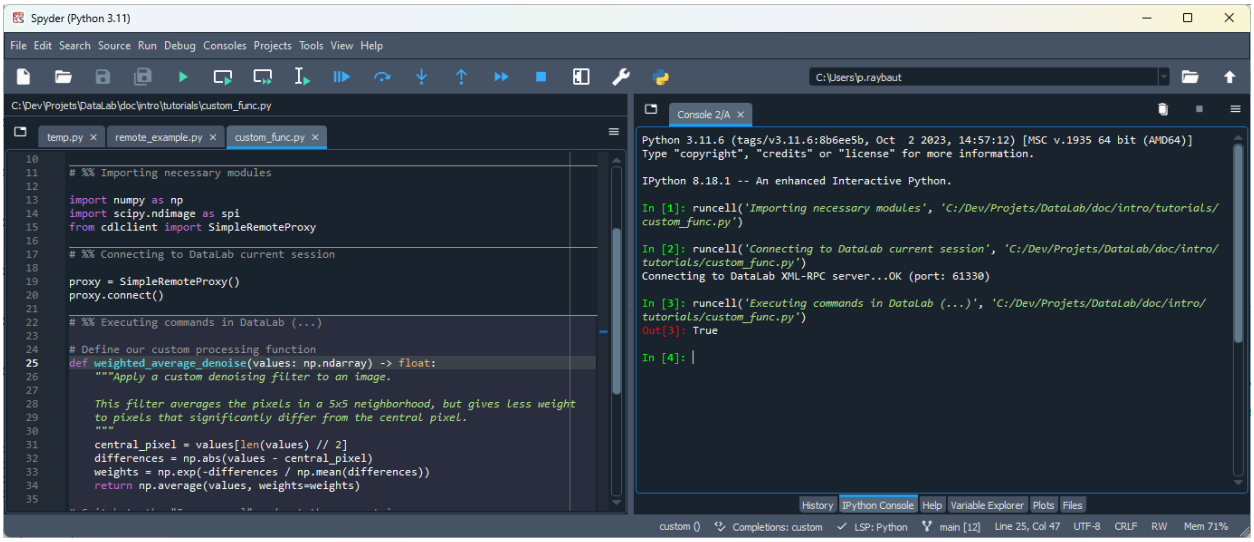


Fig. 114: The Spyder editor environment with the custom processing script.

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.

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):

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

- 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.

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)
        mean_difference = np.mean(differences)
        if mean_difference == 0:
```

(continues on next page)

(continued from previous page)

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

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


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

    FEATURE_ID = "org.datalab.examples.custom-filters:weighted-average-denoise"
    PLUGIN_INFO = datalab.plugins.PluginInfo(
        id="org.datalab.examples.custom-filters",
        name="My custom filters",
        version="1.0.0",
        description="This is an example plugin",
        capabilities=(datalab.plugins.PluginCapability.PROCESSING,),
    )

    def register_computations(self) -> None:
        """Register computations owned by this plugin."""
        wrapped_func = sipi.Wrap1to1Func(weighted_average_denoise)
        self.imagepanel.processor.register_1_to_1(
            wrapped_func,
            "Weighted average denoise",
            feature_id=self.FEATURE_ID,
            owner_plugin_id=self.plugin_id,
        )

    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        with acth.new_menu(self.PLUGIN_INFO.name):
            acth.action_for(self.FEATURE_ID)
```

To test it, we have to add the plugin script to one of the plugin directories that are discovered by DataLab at startup: configure them under "Plugins > Configure plugins… > Advanced" (see the Plugins section for more details, or the Automated detection of defects and particles on images for an example). We then reload plugins from the "Plugins" menu (or restart DataLab) and we can see that the plugin has been loaded.

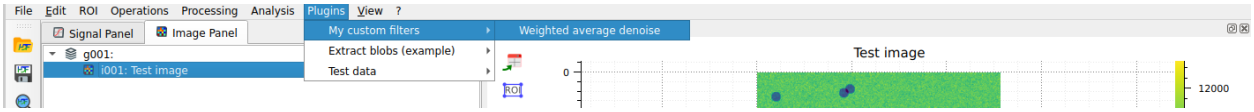


Fig. 116: After reloading plugins (or restarting DataLab), we can see that the plugin has been loaded.

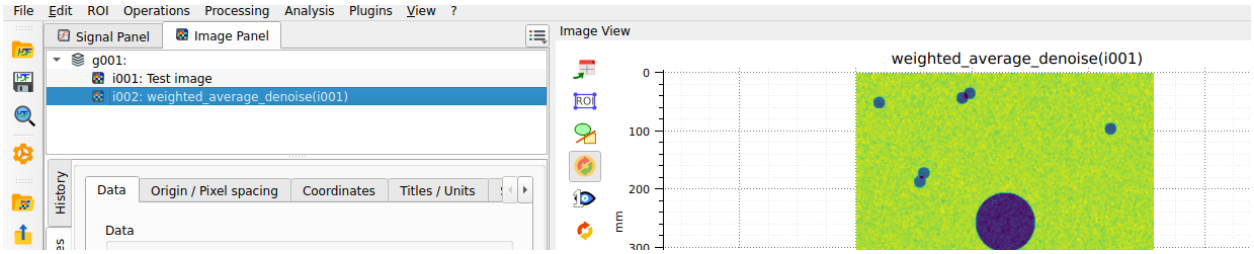


Fig. 117: We generate again our test image using (see the first steps of the tutorial), and we process it using the plugin: "Plugins > My custom filters > Weighted average denoise".

From an idea to the plugin catalog

This tutorial shows how to turn an idea into a plugin available to all DataLab users: we create a plugin in a single Python file, try it in DataLab, then publish it in the DataLab plugin catalog, from which users install it in a few clicks. No packaging project and no PyPI account are needed.

As an example, we write a "Spectrum Tools" plugin, whose processing sets the negative values of a signal to zero. Replace it with your own idea: a processing specific to your field, a file format, an analysis…

What you need

- DataLab installed as a Python package, for example with `pip install datalab-platform` (see Installation): the `datalab-plugin` command used below is not provided by the standalone version of DataLab;
- a GitHub account, to publish the source code of the plugin and to propose it to the catalog.

Step 1: create the plugin

Create the plugin with the `datalab-plugin` command. The `--github-account` option derives the plugin ID from your GitHub account, here `alice`, as expected by the catalog:

```
datalab-plugin create --name "Spectrum Tools" --description "Spectral processing␣
↪tools for DataLab" --github-account alice
```

The command writes the `datalab_spectrum_tools.py` file to the user plugin directory (see Where to put a plugin?) and prints its location. The beginning of the file declares the identity of the plugin:

```
PLUGIN_DESCRIPTION = "Spectral processing tools for DataLab"
PLUGIN_ID = "io.github.alice.spectrum-tools"
PLUGIN_NAME = "Spectrum Tools"
```

Its version, `0.1.0`, is given by the `PLUGIN_INFO` attribute of the plugin class. Keep the plugin ID unchanged once the plugin is published: DataLab and the catalog use it to identify the plugin.

Step 2: try and develop the plugin

Start DataLab, or select Plugins > Reload plugins if it is already running. In the signal panel, the Plugins menu now holds a Spectrum Tools submenu, with a Sample processing action. Create a signal with Create > Sine, keep it selected, and run this action: it adds a new signal, whose negative values are set to zero.

Open the file in your editor. The processing is the `sample_processing` function, which receives the Y values of the selected signal as a NumPy array and returns new ones:

```
def sample_processing(y: np.ndarray) -> np.ndarray:
    """Return new Y values computed from those of the selected signal.

    Edit this function to implement your own processing.
    """
    return np.maximum(y, 0.0)
```

Implement your own processing, give the action a meaningful title in the `register_computations` method, then select Plugins > Reload plugins to try your changes, without restarting DataLab. To go further (processing parameters, images, file formats…), see Developing plugins, and take the plugins of the catalog as examples (see Plugins of the catalog).

Step 3: share the file with your colleagues

The plugin can already be used by others: send them the file. They install it with Plugins > Configure plugins… > From file (see From file tab), including in the standalone version of DataLab.

Step 4: publish the source code on GitHub

Create a public GitHub repository, for example `datalab-spectrum-tools`, holding:

- the `datalab_spectrum_tools.py` file;
- a `LICENSE` file, with an OSI-approved open-source license, such as BSD-3-Clause or MIT (the contribution guide of the catalog lists the accepted licenses);
- a `README.md` file, describing what the plugin does and how to use it.

Then create a release of the repository, tagged `v0.1.0` after the version of the plugin, and attach the unchanged `datalab_spectrum_tools.py` file to it. With the GitHub CLI, from the folder of the repository:

```
gh release create v0.1.0 datalab_spectrum_tools.py --title "Spectrum Tools 0.1.0" --
↪notes "First release"
```

Keep the same file name for all versions: do not add the version number to it.

Step 5: propose the plugin to the catalog

Fork the catalog repository and add a `plugins/io.github.alice.spectrum-tools.yaml` file, named after the plugin ID:

```
id: io.github.alice.spectrum-tools
name: Spectrum Tools
kind: module
summary: Spectral processing tools for DataLab
license: BSD-3-Clause
repository: https://github.com/alice/datalab-spectrum-tools
capabilities: [processing]
keywords: [spectroscopy, spectrum]
releases:
  - version: "0.1.0"
    filename: datalab_spectrum_tools.py
    sha256: <SHA-256 digest of the released file>
    requires_python: ">=3.10"
```

`kind: module` means that the plugin is a single Python file: the catalog offers such plugins to DataLab Desktop. The ID and version must agree with those of the file; the catalog shows the name given in the YAML file. The `sha256` field gives

the digest of the file attached to the release; you may compute it with `sha256sum datalab_spectrum_tools.py` on Linux and macOS, or with `Get-FileHash datalab_spectrum_tools.py` in Windows PowerShell (in lowercase). If you leave it out, the first run of the automatic checks fails and prints the line to add.

Then open a pull request. Its automatic checks download the file from the release, verify its digest and inspect it without running it: file name, Python syntax, plugin identity and version, license. The maintainers of the catalog then check that the plugin matches its description; this review is not a security audit.

Step 6: your plugin is in the catalog

Once the pull request is merged, the catalog is published again: the plugin appears on the DataLab plugin catalog page, marked Community, and in the Catalog tab of the Plugins > Configure plugins… dialog, from which DataLab users install it with one click (see Catalog tab).

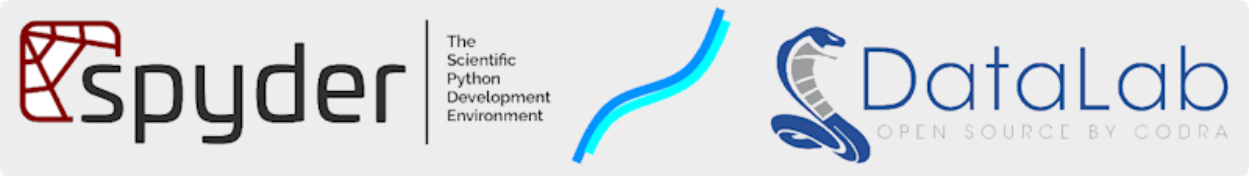
To publish a new version, increase the version in the file, create a new release (for example `v0.2.0`) holding the file under the same name, then add this version, with its digest, at the end of the `releases` list of the YAML file, in a new pull request. The Catalog tab then offers users to update the plugin.

> **Larger plugins**
>
> For a plugin made of several modules, with resources, tests or dependencies, create a layered project with `datalab-plugin create --project` (see Creating a layered plugin project). Its release workflow builds the wheel of the plugin when you push a version tag, attaches it to a GitHub release with its SHA-256 digest, and prints the entry to submit to the catalog. A wheel may also declare a distinct DataLab-Web plugin class in the `datalab.web_plugins` entry-point group, so that the plugin is offered to the users of both editions (see Plugins and DataLab-Web).

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()

# %% 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
```

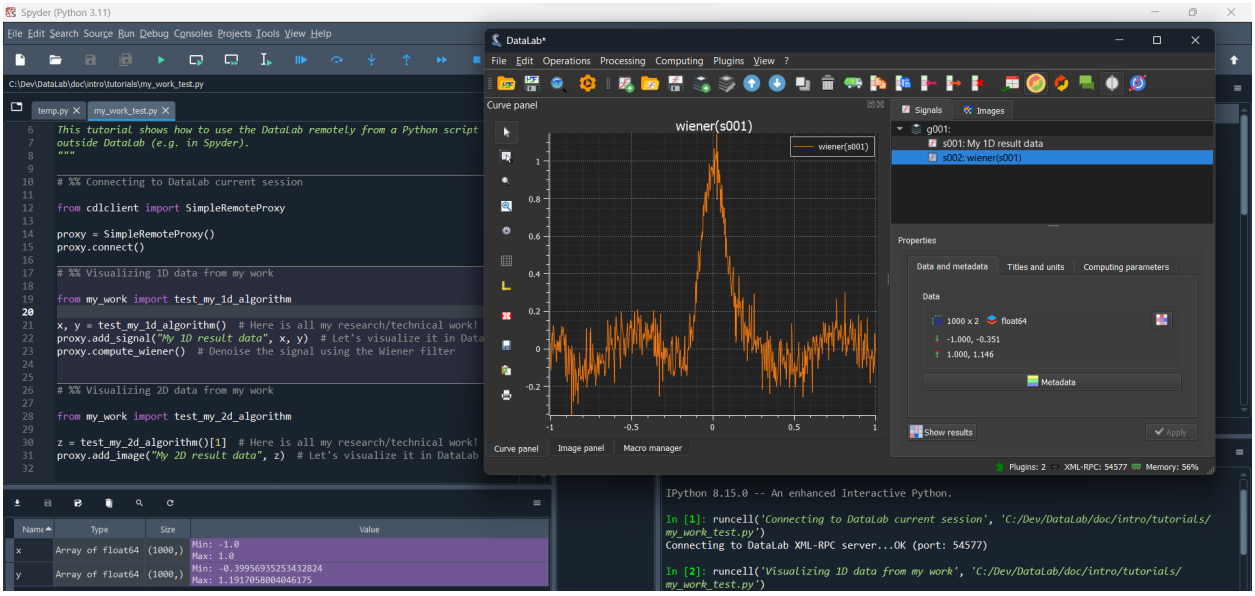


Fig. 118: In this screenshot, we can see the result of evaluating the first two cells: the first cell connects to DataLab, and the second cell visualizes the 1D data returned by the `test_my_1d_algorithm` function.

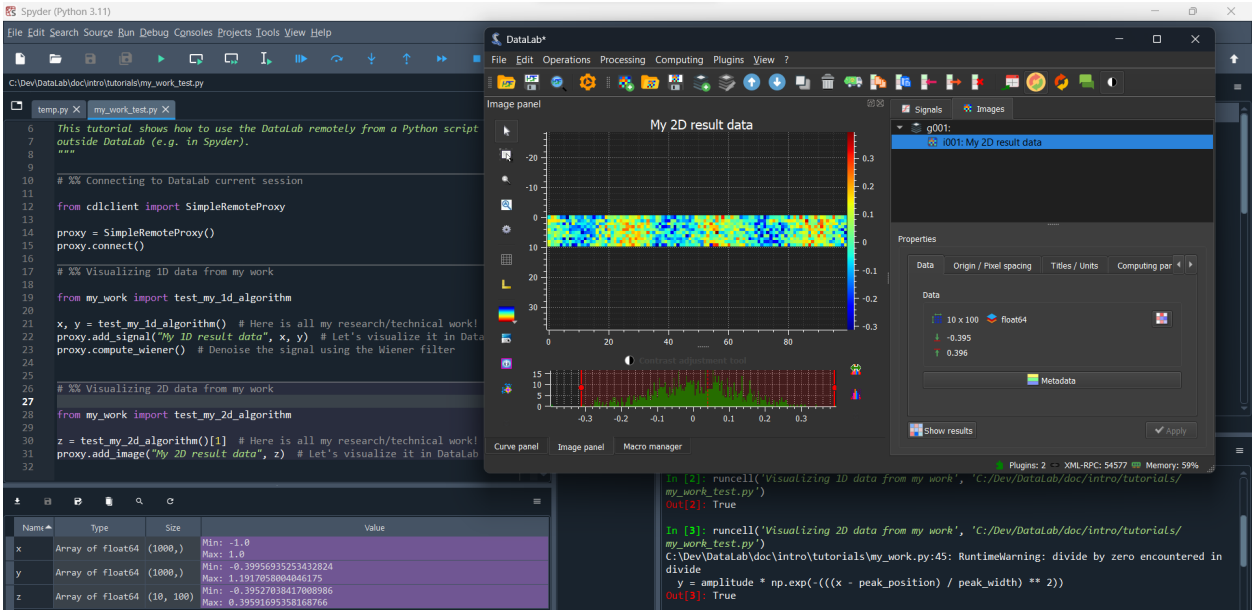


Fig. 119: In this screenshot, we can see the result of evaluating the third cell: the `test_my_2d_algorithm` function returns a 2D array, which we can visualize directly in DataLab.

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()
    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 * max(i**2, 1)
        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)
    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 test_my_2d_algorithm

x, z = test_my_2d_algorithm(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.

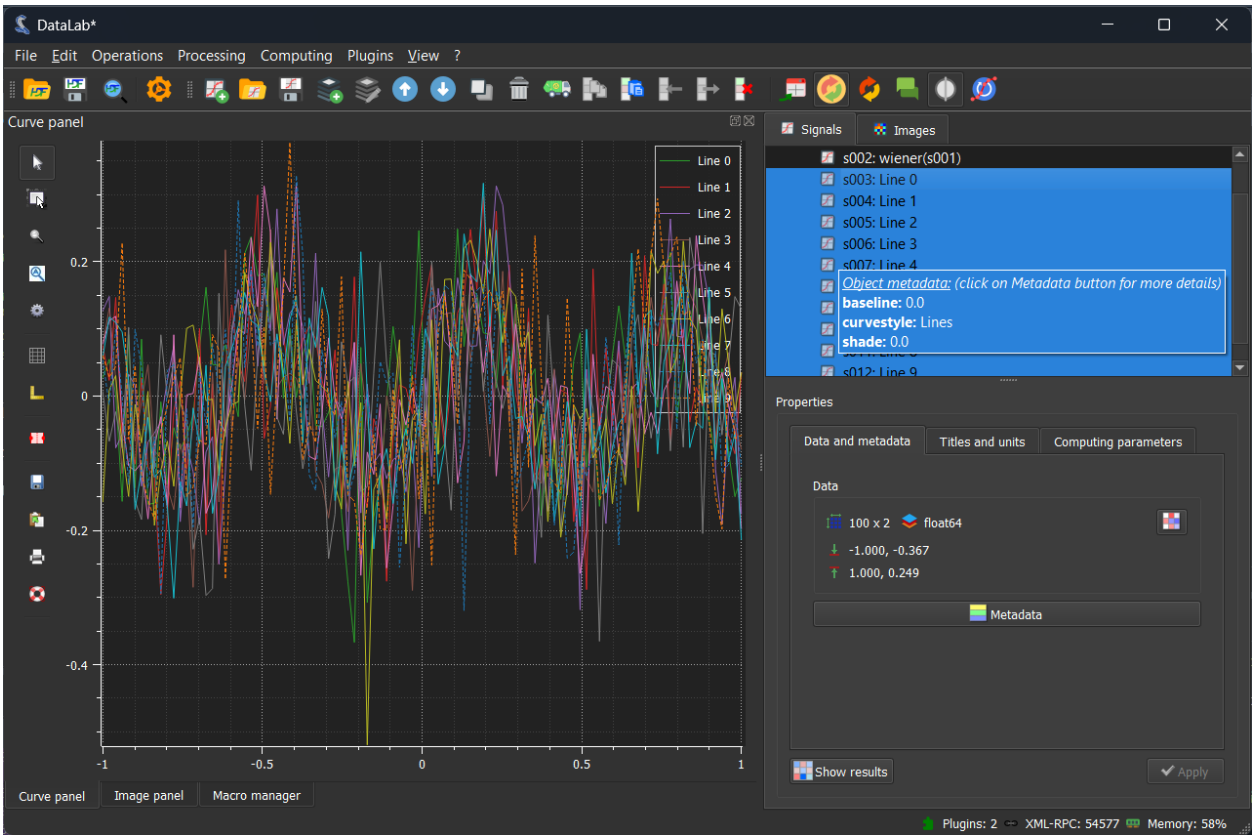


Fig. 120: In this screenshot, we can see the result of evaluating the first cell: the `test_my_2d_algorithm` function is called with the `debug_with_datalab` parameter set to `True`: 10 1D arrays are generated and visualized in DataLab.

> **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

USE CASES

DataLab is used in research and industry to analyze data produced by scientific instruments – cameras, digitizers, spectrometers – in a single workspace, without having to develop a custom application. It was initially created for the plasma diagnostics of CEA's Laser Megajoule facility, where it is used in production, and has since been applied to many other fields.

The pages below show how DataLab addresses a concrete problem in each domain, with demonstration data, step-by-step tutorials and, for complete workflows, application plugins of the DataLab plugin catalog.

2.1 DataLab for spectroscopy

2.1.1 The problem

Spectra coming out of an instrument – Raman, FTIR, XRF, or any energy/wavelength scan – rarely come clean: noise, baseline drift, overlapping peaks. The usual options are vendor software (closed, one instrument), spreadsheets (error-prone, no traceability), or custom scripts (weeks of work, hard to share with non-programmers).

2.1.2 What DataLab does

DataLab lets you chain the whole analysis interactively. Enable History recording to keep processing steps and their parameters:

- filter noise (Wiener, moving average, and more),
- correct the baseline, including linear detrending restricted to peak-free regions – a robust approach when peaks dominate the signal,
- select a region of interest and fit a model (Gaussian, Lorentzian, Voigt, multi-peak…),
- save the whole workspace (data + processing history + metadata) to a single HDF5 file that colleagues can reopen. DataLab-Web can open the data, but does not preserve the desktop History when it saves the workspace again.

Every operation is also available from Python (scripts, Jupyter, macros), so an interactive analysis can be turned into an automated pipeline processing a whole series of spectra.

2.1.3 Proof in production

DataLab was built for and is used in production at CEA's Laser Megajoule facility, where spectrometer data are processed and visualized as part of the plasma diagnostics workflows.

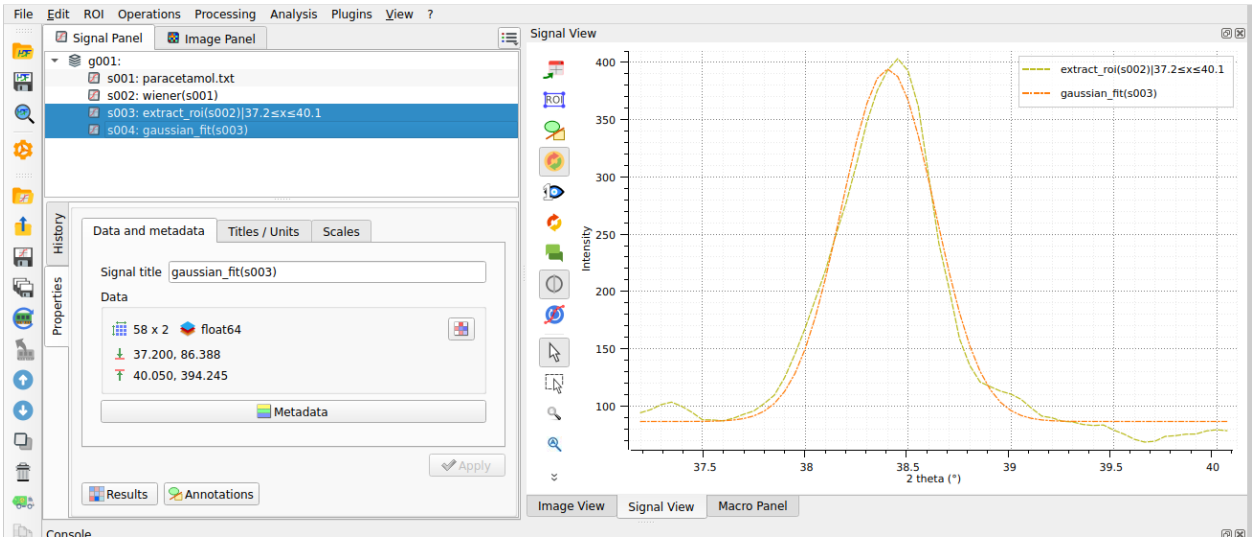


Fig. 1: A Gaussian fit on a region of interest of a spectrum in DataLab.

2.1.4 Try it

- No install: open DataLab-Web with a demo spectrum already loaded in your browser – your data never leaves your machine.
- Step-by-step tutorial: Spectrum analysis: denoising, baseline correction and peak fitting.
- Or install the desktop application.

2.2 DataLab for photonics and lasers

2.2.1 The problem

Aligning a laser bench or characterizing an optical setup means extracting quantitative measurements from camera images: beam size at several positions along the propagation axis, fringe radii on an interferogram, intensity profiles. Doing this with vendor beam-profiling software locks you to one camera; doing it with scripts means rebuilding ROI handling, calibration and visualization from scratch.

2.2.2 What DataLab does

DataLab works on images and the signals extracted from them, in the same workspace:

- load a whole folder of camera images at once,
- threshold, define regions of interest, apply linear calibration,
- extract line or radial intensity profiles and fit them (Gaussian and other models) to get FWHM and beam size,
- detect contours and fit them to circles to measure interferogram fringe radii,
- repeat the same analysis on a stack of images and plot, e.g., the beam size as a function of the position along the propagation axis.

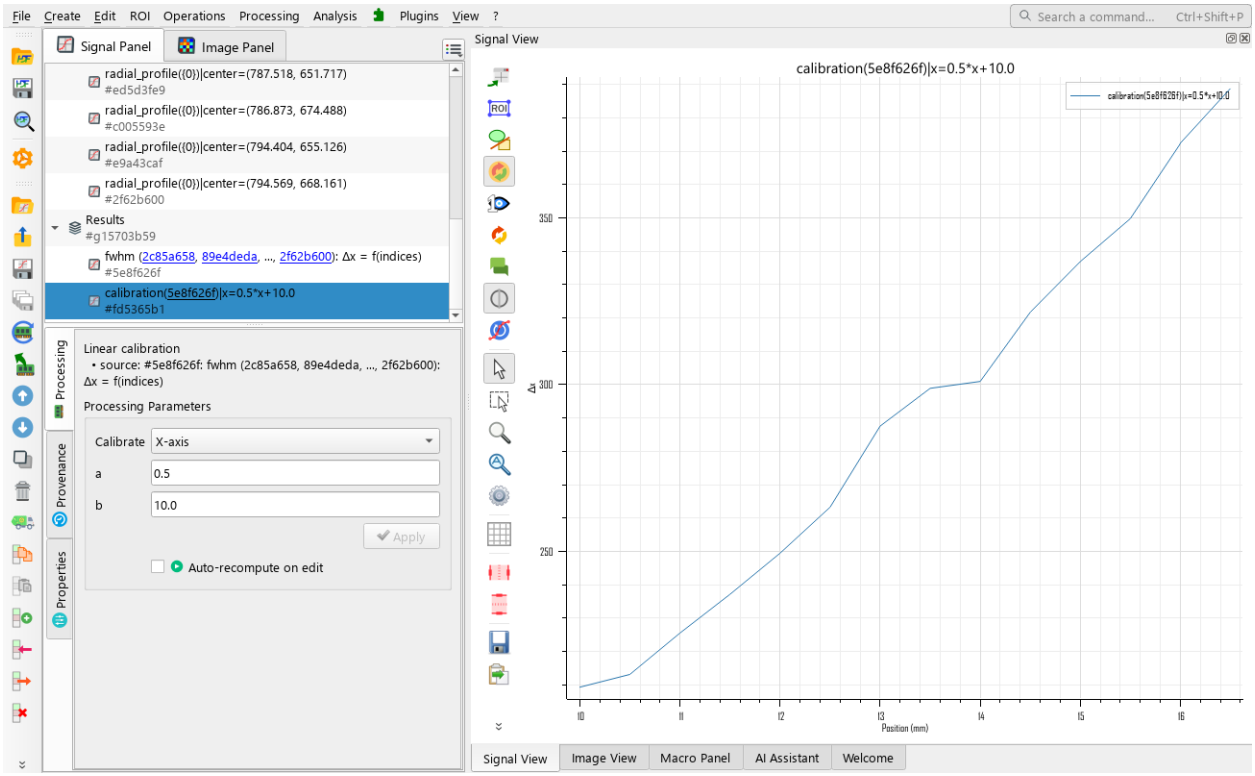


Fig. 2: Beam size along the propagation axis, computed from a stack of camera images in DataLab.

2.2.3 Proof in production

DataLab was created for the plasma diagnostics of CEA's Laser Megajoule facility – one of the world's largest laser facilities – where it processes images and signals from cameras and digitizers, and is also used for laser alignment and metrology R&D activities around the facility.

2.2.4 Try it

- No install: open DataLab-Web with demo beam and interferogram images already loaded in your browser – your data never leaves your machine.
- Step-by-step tutorials: Laser beam profiling and Analyzing Fabry-Perot interferograms.
- Or install the desktop application.

2.3 DataLab for non-destructive testing

2.3.1 The problem

Inspection produces images – radiographs, thermograms, ultrasonic C-scans, surface pictures – on which defects or indications must be found, counted and measured, often on whole series of parts. Manual review does not scale and is hard to make reproducible; fully custom software is expensive to build and maintain for each inspection bench.

2.3.2 What DataLab does

DataLab provides the image-analysis building blocks and keeps the whole chain traceable:

- denoise inspection images (median, binning, and other filters),
- detect blob-like features – defects, particles, spots – automatically, with several detection algorithms,
- restrict the analysis to regions of interest, get measurable results (positions, sizes) exportable to your reports,
- save the workspace (images + processing history + results) to a single HDF5 file for traceability,
- and when the built-in algorithms are not enough, plug in your own Python processing with the plugin system – your operators get a new menu entry, not a new software.

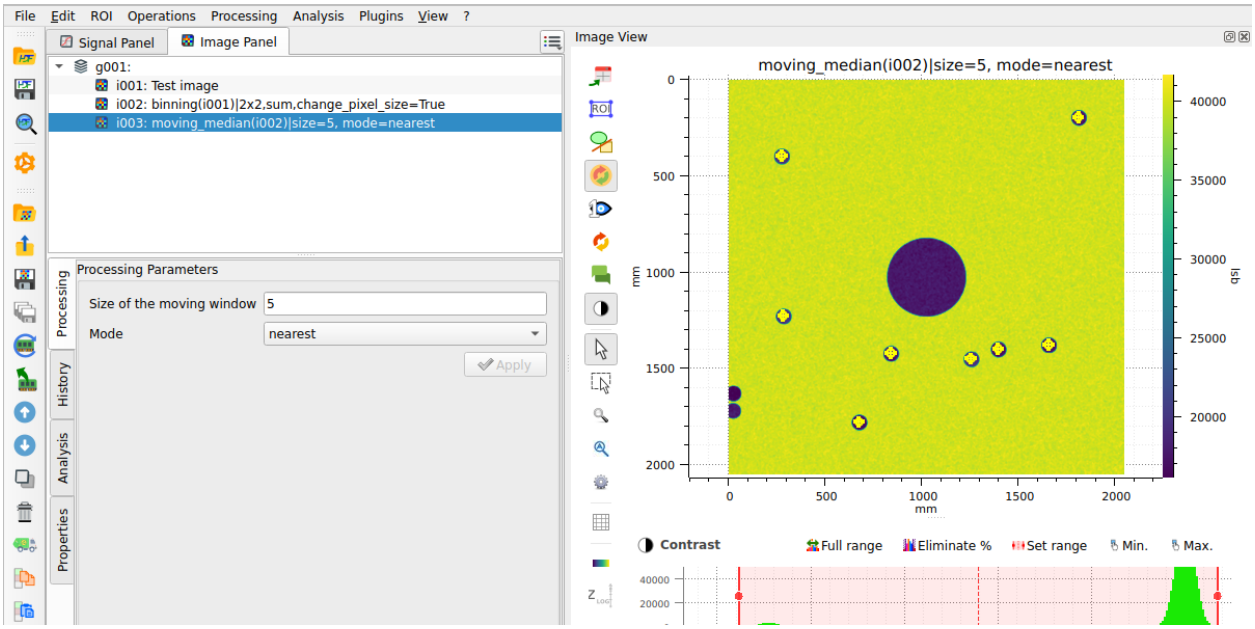


Fig. 3: Automatic blob detection on a denoised test image in DataLab.

2.3.3 X-ray radiography

For X-ray radiography, the X-ray Radiography & NDT plugin adds dedicated features to DataLab:

- radiographic display: gray scale and automatic window, film convention, windows from regions of interest or DICOM attributes,
- calibrated measurements: several named magnifications per image, one per object plane, and distances, angles, diameters and areas measured in the detector plane and in each object plane at once,
- detector corrections (offset, gain, defective pixels) and physical quantities: attenuation, equivalent thickness, step wedge thickness calibration with beam hardening,
- enhancement filters with sizes in mm, and image quality: SNR, normalized SNR, CNR, basic spatial resolution with a duplex image quality indicator,
- NDT measurements: wall thickness in tangential radiography, detection and sizing of indications,
- pixel spacing, magnification and window of DICOM and DICONDE files,
- a live X-ray radiography simulator with known truth, and simulated and real-data examples.

The plugin is an official plugin of the DataLab plugin catalog, at the Alpha stage: its methods are validated on synthetic data with known truth and checked on open real data. They take inspiration from radiography standards (ISO 17636-2, ISO 19232, ASTM E2597, ISO 20769) without claiming compliance.

2.3.4 Proof in production

In the field of non-destructive testing, CEA entrusted CODRA with X-GRID, a software for the automatic reconstruction of radiographic scenes from partial X-ray images – with no prior metadata on position, orientation or magnification. Its processing pipeline (denoising, robust blob detection, homography estimation, image fusion) was prototyped interactively with DataLab before being integrated into the production tool. This work was presented at EuroSciPy 2026.

2.3.5 Try it

- No install: open DataLab-Web with a demo inspection image already loaded in your browser – your data never leaves your machine.
- Step-by-step tutorial: Automated detection of defects and particles on images.
- X-ray radiography: install the X-ray Radiography & NDT plugin from the Catalog tab of Plugins > Configure plugins… (see Catalog tab), then open its simulated weld example from Plugins > X-ray Radiography & NDT > Examples; see its documentation.
- Or install the desktop application.

2.4 DataLab for scientific camera characterization

2.4.1 The problem

Before trusting the images of a scientific camera or detector, you need to know how it behaves in your own conditions: its response to light, its temporal noise, the non-uniformity of its pixels, its conversion gain and read noise, its dark current and its defective pixels. Datasheets give typical values, not those of your unit at your settings; ad hoc scripts take time to write and validate, and their results are hard to compare from one camera, one setting or one campaign to the next.

2.4.2 What DataLab does

The Camera & Detector Characterization plugin adds complete characterization methods to DataLab, run on the dark and flat frames of your workspace:

- relative characterization in digital numbers (DN): response curve, temporal noise, DSNU-like and PRNU-like maps, row and column profiles, candidate defective pixels,
- photon transfer curve: conversion gain (e^-/DN), read noise, saturation capacity and dynamic range,
- dark current and hot pixels, from a dark ramp,
- results added to the workspace as signals, images and metrics tables, next to the frames they come from, ready for the processing and analysis features of DataLab and saved with them in the workspace,
- a scientific camera simulator, with a live view, which acquires dark and flat frames ready for the methods,
- the methods and their examples are presented in the Applications panel, which tells whether the selected frames can be analyzed, and why not.

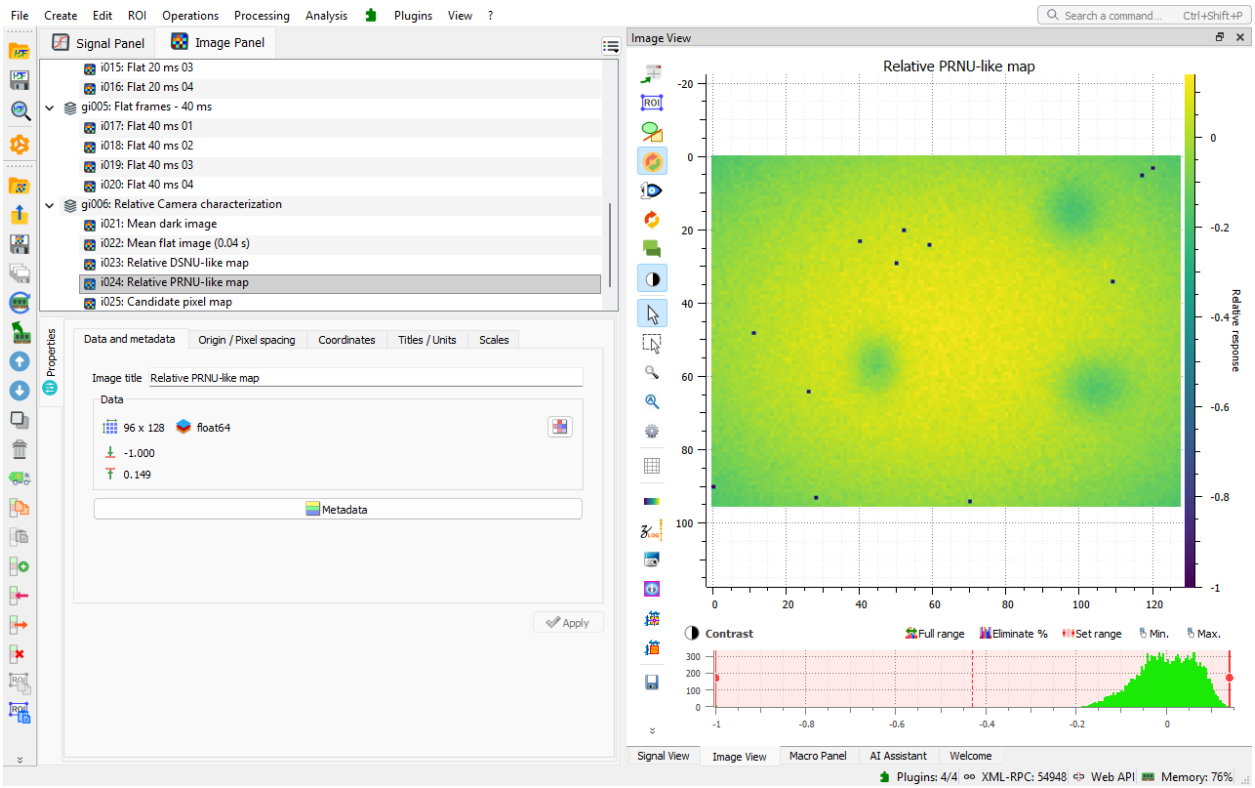


Fig. 4: Relative PRNU-like map computed from the quickstart campaign of the Camera & Detector Characterization plugin in DataLab.

2.4.3 Status

The plugin is an official plugin of the DataLab plugin catalog, for DataLab Desktop and DataLab-Web, at the Alpha stage: its methods are validated on synthetic data with known truth, and do not claim compliance with the EMVA 1288 standard.

2.4.4 Try it

- No install: open DataLab-Web with the quickstart campaign loaded and the camera characterization method selected in your browser – your data never leaves your machine.
- In DataLab Desktop: install the plugin from the Catalog tab of Plugins > Configure plugins… (see Catalog tab), then click its tile on the Welcome page.
- Plugin documentation: getting started and methods.
- Or install the desktop application.

2.5 DataLab for transient and pulsed signals

2.5.1 The problem

Pulsed lasers, pulse generators, amplifiers, detectors and accelerators are characterized from transient signals recorded by oscilloscopes and digitizers, often over hundreds of shots: which shots are valid, what the representative pulse looks like, how much the timing jitters and the amplitude drifts, how fast a system responds to a step, what the delay between two channels is. Oscilloscope measurements describe one shot at a time, and campaign scripts must handle baselines, polarities, invalid shots and alignment before giving any number.

2.5.2 What DataLab does

DataLab analyzes transient signals out of the box:

- Analysis > Extract pulse features… gives the amplitude, rise and fall times, FWHM and timing of step and square pulses (see Analysis features on Signals),
- Processing > Stability analysis gives the Allan, Hadamard and total variances and the Allan and time deviations of a time series (see Processing Signals).

The Pulse & Transient Characterization plugin adds complete methods for series of shots:

- pulse campaign: per-shot features and quality flags, raw and aligned mean pulses,
- shot-to-shot stability: timing jitter, drifts, amplitude and energy stability, Allan deviation, regime changes,
- step response: rise time, overshoot, settling time, ringing and bandwidth,
- delay between two channels, with relative and common-mode jitter,
- pulse-height spectrum: energy calibration and resolution of a detector,
- an oscilloscope simulator, with a live view, which acquires laser pulses, amplifier steps or delayed pulse pairs ready for the methods.

Results are added to the workspace as signals and metrics tables, and the methods and their examples are presented in the Applications panel.

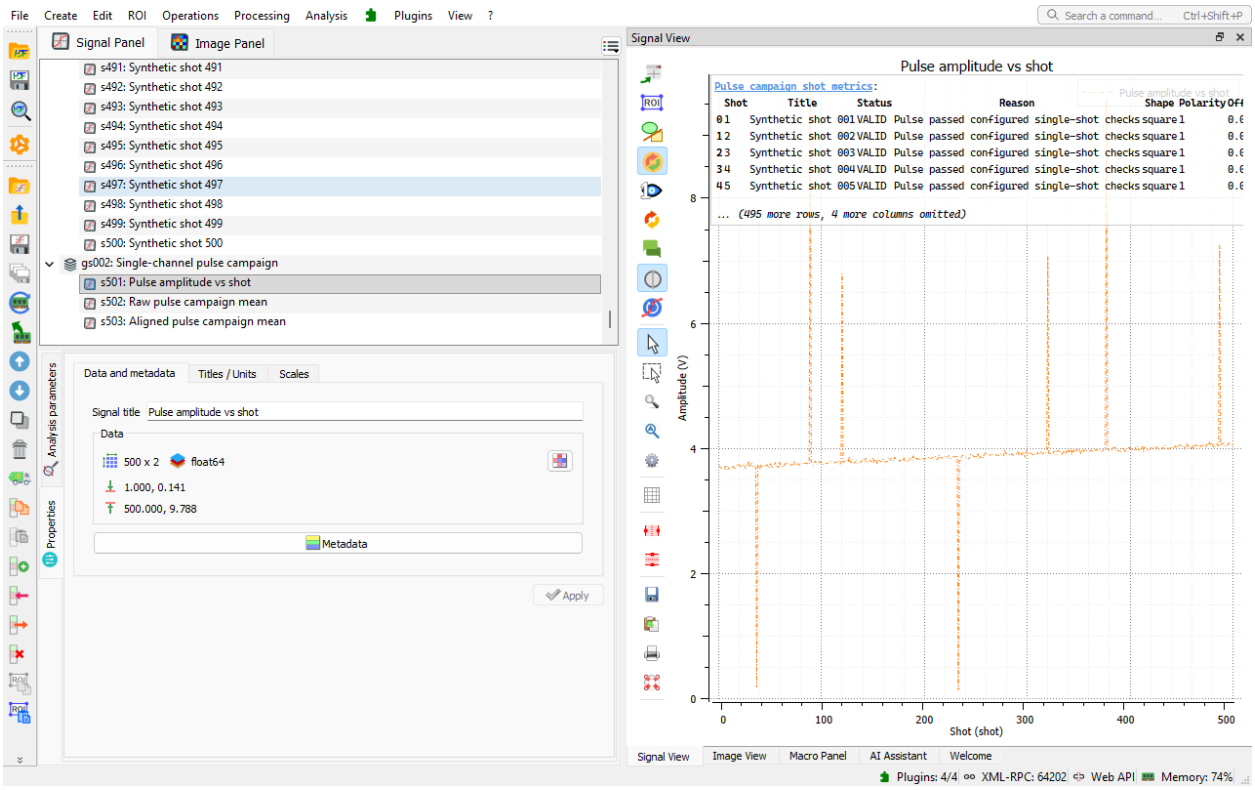


Fig. 5: Amplitude versus shot, with its per-shot metrics table, computed from the 500-shot demonstration campaign of the Pulse & Transient Characterization plugin in DataLab.

2.5.3 Status

The plugin is an official plugin of the DataLab plugin catalog, for DataLab Desktop and DataLab-Web, at the Alpha stage: its methods are validated on synthetic data with known truth, and do not claim compliance with a standard.

2.5.4 Try it

- No install: open DataLab-Web with a 500-shot campaign and the pulse campaign method selected in your browser – your data never leaves your machine.
- In DataLab Desktop: install the plugin from the Catalog tab of Plugins > Configure plugins… (see Catalog tab), then click its tile on the Welcome page.
- Plugin documentation: getting started and methods.
- Or install the desktop application.

2.6 DataLab for instrument control and data acquisition

2.6.1 The problem

On a test bench or a facility, data usually go from the instrument to its vendor software, then to files, then to the analysis tool: there is no live link between what the instrument measures and what you analyze, and the acquisition context – which device, which setting, when – is easily lost along the way. Writing a dedicated acquisition application for each instrument, with its own display and analysis, is expensive.

2.6.2 What DataLab does

A DataLab plugin may provide instruments: the plugin describes the instrument settings and how to read or control the device, and DataLab provides the user interface (see Instruments):

- a window with a live view of the instrument data (signals or image) next to its settings, refreshed when the settings change or continuously in Live mode,
- Acquire adds the acquired signals or images to the workspace, in a new group or in the current one, selected and ready for the processing, analysis and application methods of DataLab,
- the instrument is called outside the user interface thread: a slow or unreachable device does not freeze DataLab, and it is released when the window closes,
- the plugin may record the origin of each acquired object (device, settings, time) in its metadata, which is saved with the workspace.

The TANGO Instruments example plugin shows this approach with the devices of the TANGO control system, used by many large research facilities:

- a device attribute is shown live, a spectrum as a signal, an image as an image, with the units and axis calibrations given by a JSON device profile, which holds the site-specific values (device name, calibrations, commands),
- Acquire copies the last received value, runs the trigger command of the profile and takes the value it measured, or takes a series of the next received values,
- each acquired object records its origin (device, attribute, times, quality, profile and trigger): a saved workspace reopens without the device nor the plugin,
- a simulated oscilloscope and a simulated camera let you try it without hardware: the Simulated TANGO devices… tool serves them with PyTango on your computer while its window is open, without any command line, and stops them when the window closes.

The simulators of the camera and pulse characterization plugins are instruments of the same kind. Acquisition software may also send its data to DataLab, with the remote control (XML-RPC) or the Web API (HTTP/JSON).

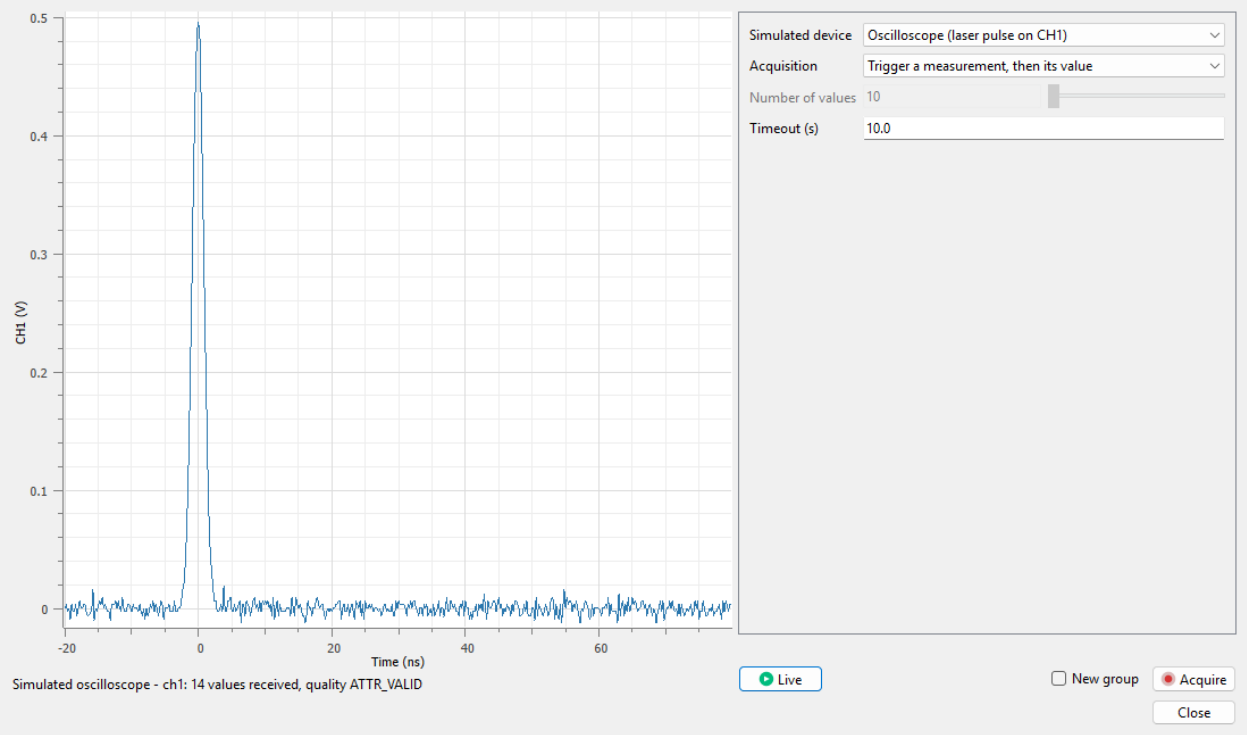


Fig. 6: Live view of the simulated oscilloscope in the window of the Simulated TANGO devices… tool of the TANGO Instruments plugin in DataLab.

2.6.3 Status

TANGO Instruments is an official example plugin of the DataLab plugin catalog: it is a demonstration, not a finished or supported feature, and has been validated with simulated devices only. It requires PyTango in the Python environment of DataLab: the standalone application and DataLab-Web, which cannot reach TANGO devices, are not supported.

2.6.4 Try it

- In DataLab: install PyTango in the Python environment of DataLab (e.g. `pip install "pytango>=9.5.1"`), then the plugin from the Catalog tab of Plugins > Configure plugins… (see Catalog tab).
- Without hardware: open Simulated TANGO devices… from the Applications panel or from the plugin's submenu of the Plugins menu, choose the simulated oscilloscope or camera, then click Acquire. To serve the same devices to another TANGO client, run `python -m datalab_tango.simulation profiles` in this Python environment.
- Plugin documentation: device profiles, acquisition modes and simulated devices.
- Your own instruments: see Instruments and From an idea to the plugin catalog.

CHAPTER

THREE

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.

3.1 Overview & Common features

3.1.1 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 contains the Signal Panel and Image Panel tabs, which list the data sets currently loaded in each panel. Clicking a tab 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.

In the list of data sets, the signal, image or group under the mouse cursor is highlighted, and a  delete button appears at its right. Clicking this button deletes the item, or the whole selection if the item is part of it, after confirmation. Deleting an item which is not selected keeps the current selection unchanged. Like the “Remove” action of the “Edit” menu, the deletion is recorded in the History Panel. The button may be hidden in the Settings (“Visualization” tab, “Common” sub-tab).

Additional panels are available as dockable widgets, and may be shown or hidden from the “View” menu: the Welcome page, the Applications panel, the History Panel, the Macro panel (see Macros) and the AI Assistant.

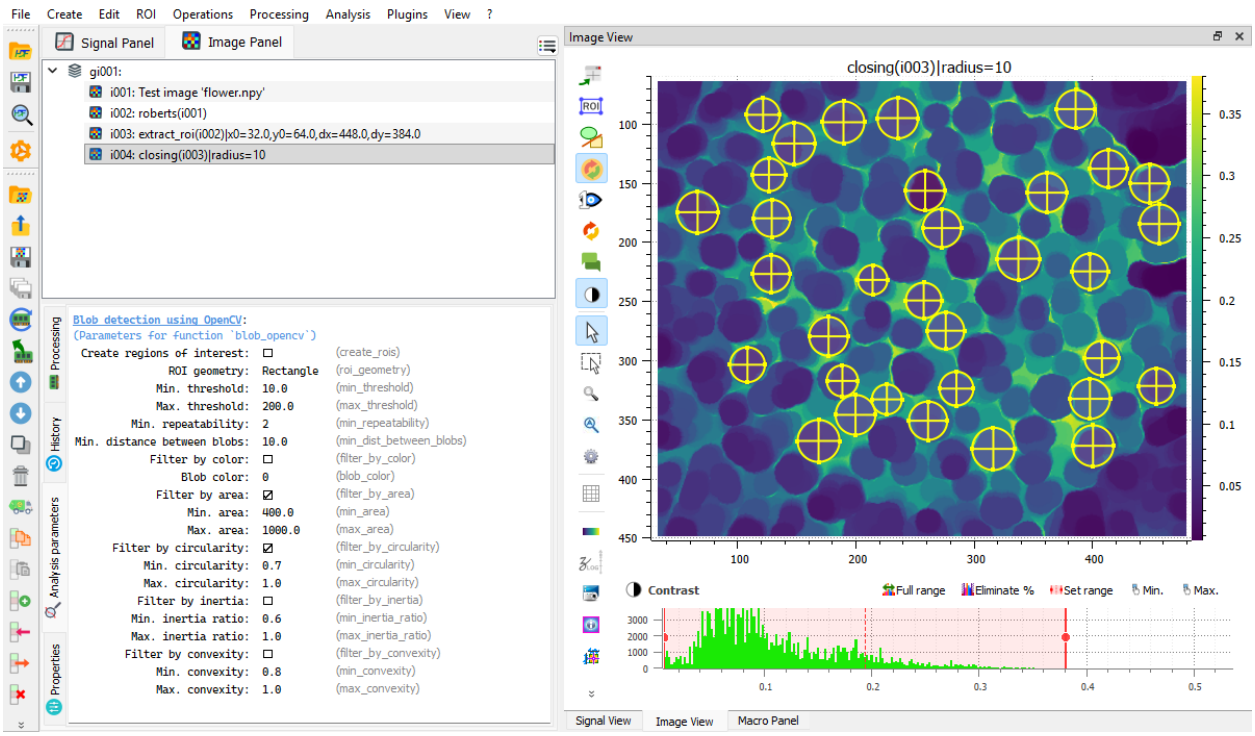


Fig. 1: DataLab main window

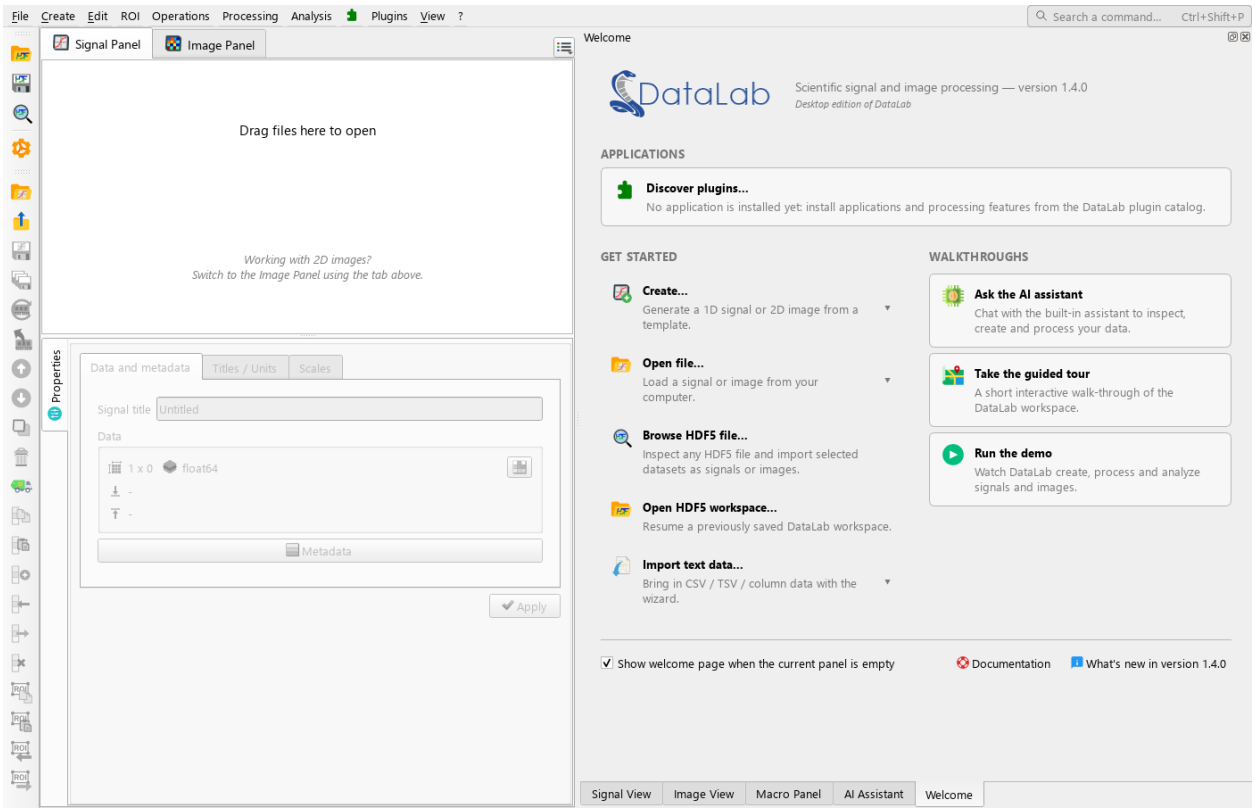


Fig. 2: DataLab main window, at startup.

Welcome page

A Welcome page is shown next to the signal and image views when DataLab starts, and whenever the current signal or image panel is empty (for example after switching to an empty panel or deleting its last object). It gathers the main actions to get started:

- create a signal or an image from a template, open signal or image files, browse an HDF5 file, open a previously saved HDF5 workspace, or import text data (CSV, TSV, …) with the import wizard. When needed, a menu lets you choose between signals and images;
- ask the AI Assistant, take the guided tour, or run the demo.

Links at the bottom of the page open the online documentation and the release notes of the running version.

An Applications section at the top of the page gathers the application plugins. While no application is installed, it invites you to discover plugins: "Discover plugins…" opens the Catalog tab of the plugin configuration dialog, which lists the plugins of the DataLab plugin catalog, ready to install (see Plugins). This section is hidden when third-party plugins are disabled.

When application plugins are installed, the Applications section shows their tiles, and its "Discover plugins…" button still opens the plugin catalog: clicking the main tile of an application shows its page in the Applications panel, or starts the main action provided by the plugin. An application may provide other tiles, for example to open an example directly: they are shown next to its main tile when the section has room for them, and otherwise move to the menu of the main tile. This menu, opened from the "…" button of the main tile or by right-clicking it, also lets you pin the application to the top of the section or hide it from the welcome page. Pinned applications come first, then recently used ones, then the others by name. The section is limited to two rows of tiles by default (this number may be changed in the Settings): when more applications are installed, the last tile gives the number of applications left out and shows the Applications panel. The "Browse all applications…" button also shows this panel, where the "…" menu of each application offers the same "Show on welcome page" and "Pin to the welcome page" options (see Plugins).

The Welcome page is a regular dockable panel: as soon as a signal or an image is created or opened, the corresponding view is brought to the front, and the Welcome page remains available as a tab. It may be closed like any other panel, and reopened at any time from the "Welcome page" entry of the "?" (Help) menu or from the "View" menu; it is also reopened automatically when the current panel becomes empty. The "Show welcome page when the current panel is empty" check box, at the bottom of the page, controls this automatic display, including at startup (this option is also available in the Settings).

Applications panel

The Applications panel presents the methods of the installed application plugins. It is shown by the Applications entry of the menu bar, by the application tiles of the Welcome page, or from the "View" menu. The first time, it floats beside the main window, as high as this window, so that you keep your data and their views in sight while using it. You may then dock it like any other panel, for example on the left of the main window, next to the list of data sets whose selection it analyzes, tabify it with another panel, or leave it floating: DataLab shows it again where you left it, with the same size. When the panel is requested while already shown, without switching to another application, it flashes briefly to show where it is.

When several application plugins are installed, their number is given at the top of the panel, above a list which selects the application: click it to show all the applications, with the version and the description of each one. With only one application, its name is shown instead. The "…" button next to it opens a menu with the documentation of the application and its welcome page options: "Show on welcome page" and "Pin to the welcome page".

The methods of the application are listed one below the other, followed by its tools and datasets, if any: clicking the title of one of these sections hides its content, or shows it again, and DataLab remembers this choice for all applications. Each method takes one row. Its colored dot and the text below its title tell whether the current selection can be analyzed; the tooltip gives the reasons. When metadata are missing, the status points to "Edit > Metadata > Add metadata…", and it is updated as soon as objects are modified. "Run…" starts the method on the current selection, and asks you to assign

the selected objects to its inputs only when needed; the number of created objects and the diagnostics of the run are then shown below the row.

Below the list, in the "Details" area, the panel gives the details of the selected method: click a row, or use the `Up` and `Down` keys, to select another method. The details give the version and the description of the method, whether the current selection can be analyzed and why not, the inputs it expects (with their description, and the description and the key of the metadata they require), and the examples designed for it: "Try with this example" opens an example and runs the method with suitable parameters. The splitter between the list and the details sets their heights; DataLab remembers it, as well as the method selected for each application.

Command palette

Rather than navigating through nested menus, any command may be reached by typing part of its name or of its menu path in the command palette. It lists every command available for the currently active panel, identified by its localised menu path (e.g. "Processing › Fourier analysis › FFT").

The palette may be opened in three ways:

- with the `Ctrl+Shift+P` keyboard shortcut,
- by clicking the search box in the top-right corner of the menu bar,
- from the "Command palette…" entry at the top of the "?" (Help) menu.

Type a few characters to filter the list: the matching is fuzzy, so "fft" or "fan" both reach the "Fourier analysis" entries. Results are navigated with `Up` / `Down` and triggered with `Enter`, exactly as if the command had been selected in the menu bar. With an empty query, all commands are listed alphabetically by path.

History Panel

The History Panel is an additional dockable panel that records signal and image actions in a shared active session, while its Record mode is enabled (by default, this mode is disabled when DataLab starts). It supports replay, step-by-step replay, duplication, and compatibility diagnostics. Sessions may be saved in standalone `.dlhist` files or with the workspace in HDF5 format.

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 detail in the API section (see `sigima.objects`).

For each signal or image object, DataLab stores not only the data itself but also a set of metadata describing the data and how it should be displayed. The metadata is stored in a dictionary accessible through the object's `metadata` attribute (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 files…: load a workspace from one or more HDF5 files.
- 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 .png 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 object 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 object from the 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

Live preview before applying a processing

Standard parameter dialogs for 1-to-1 processing offer a Preview checkbox, unchecked each time the dialog opens. Enable it to display a temporary result while adjusting the parameters. With several signals, images or groups selected, choose one source in the preview selector; OK still processes the whole original selection with the common parameters.

The preview uses the complete source data, including its regions of interest, without adding objects or history entries. Invalid entries suspend updates. Warnings and errors appear inside the dialog. Typing is debounced and slider drags are throttled; large data and expensive algorithms may take longer to update. The preview shows only the result, with its own axes and units, including when an image processing returns a signal.

Starting the first preview creates a dedicated process. After a preview has finished, DataLab keeps this process ready for later preview dialogs, including when switching between signal and image processing. Closing a dialog while a preview is still running stops that process; the next preview then performs a fresh startup. Reloading plugins also discards the cached preview process.

Cancel discards the draft parameters and stops the preview. With one individually selected source, OK reuses a completed, up-to-date preview once and publishes it through the normal processing path, including history. If the preview is still running or stale, or if several objects or groups are selected, DataLab runs the normal processing instead. Analysis, multi-input operations, operations without parameters and unknown custom parameter dialogs retain their existing workflow.

Bounded numeric fields automatically offer sliders in compatible processing forms and in the Processing tab. The exact text field remains authoritative; moving its slider does not limit the precision of values entered by keyboard. Fields without two usable bounds remain text-only. Sliders alone never enable preview or automatic re-processing.

Re-processing an existing result

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 > Noise reduction > 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

Apply recomputes the result in place: the existing object is updated, without creating a new object or a new history entry, and the new execution is recorded in the provenance of the result (see Provenance of results). The parameters of the corresponding History action are left unchanged: to edit them, replay the action with Step-by-step in the History Panel. The optional Auto-recompute on edit checkbox follows the same rules, waits for slider release and valid input, and does not apply edits from a previously selected object. 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 structuring element sizes
- Educational demonstrations of parameter effects on processing results

> **Note**
>
> Processing metadata requirements:
>
> - Processing functions with parameters can be edited; processing functions without parameters (e.g., absolute value, average) offer a Reapply button
> - Every input object must still exist for re-processing (the Processing tab says why it cannot be applied otherwise)
>
> When the workspace holds a provenance record for the object, the Processing tab restores the recorded call, including 2-to-1 processing (source and operand) and n-to-1 processing (every aggregated object, in order). See Provenance of results.

Example workflow

Here's a typical workflow using interactive processing:

1. Create a test signal: Create > Gaussian
   - Default parameters: A=1.0, μ=0, σ=1 (with x from -10 to 10)
2. Adjust creation parameters: In the Creation tab, change σ to 2, click Apply

- Signal is regenerated with new width

3. Apply Gaussian filter: Processing > Noise reduction > Gaussian filter…
   - Default σ=1.0
4. Fine-tune filtering: In the Processing tab, try σ=3.0, then σ=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.

3.1.2 Settings

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

General

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

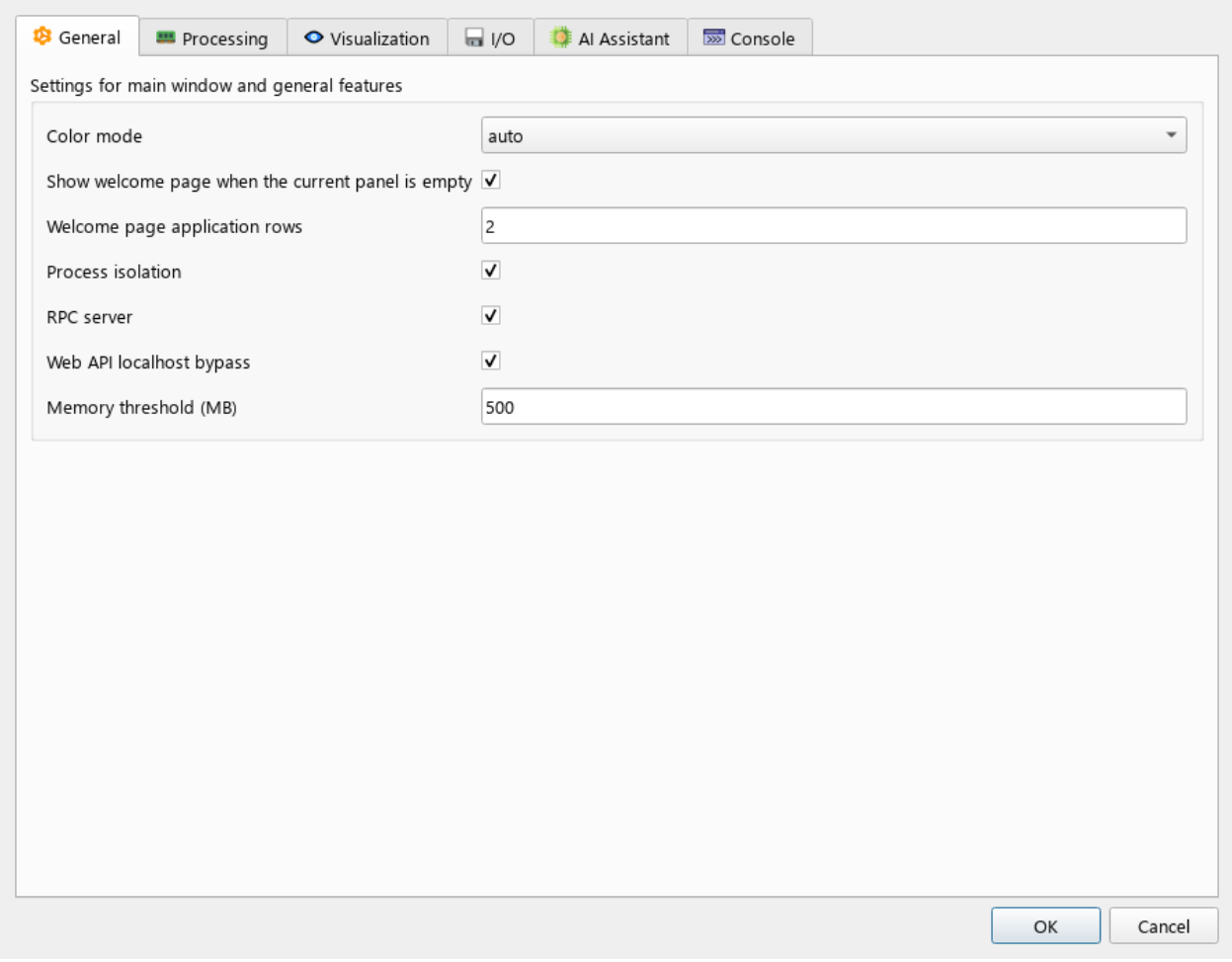


Fig. 3: General settings tab

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

Show welcome page when the current panel is empty
: When enabled (default), the welcome page is shown when DataLab starts and whenever the current signal or image panel is empty (see Welcome page).

Welcome page application rows
: Maximum number of application tile rows on the welcome page, from 1 to 10 (default: 2). Other applications remain available from the Applications panel.

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. Changing this option requires restarting DataLab.

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. Changing this option requires restarting DataLab.

Web API localhost bypass
: When enabled (default), connections from localhost (127.0.0.1) to the Web API do not require authentication. This simplifies notebook integration when DataLab-Kernel runs on the same machine. Disable for stricter security if needed.

> **Note**
>
> When this option is enabled, any application running on your local machine and any web page open in your browser can access the DataLab Web API without a token. Disable this option if you need stricter security.

Memory threshold
: Set a threshold (in MB, default: 500) 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.

> **Note**
>
> Plugin-related settings (global enabling of third-party plugins, per-plugin enabling and additional plugin directories) are not part of this dialog: they are grouped in the plugin configuration dialog, available from the `Plugins > Configure plugins...` menu action (see Plugins).

Processing

The Processing settings tab controls computation behavior and default parameters:

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

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

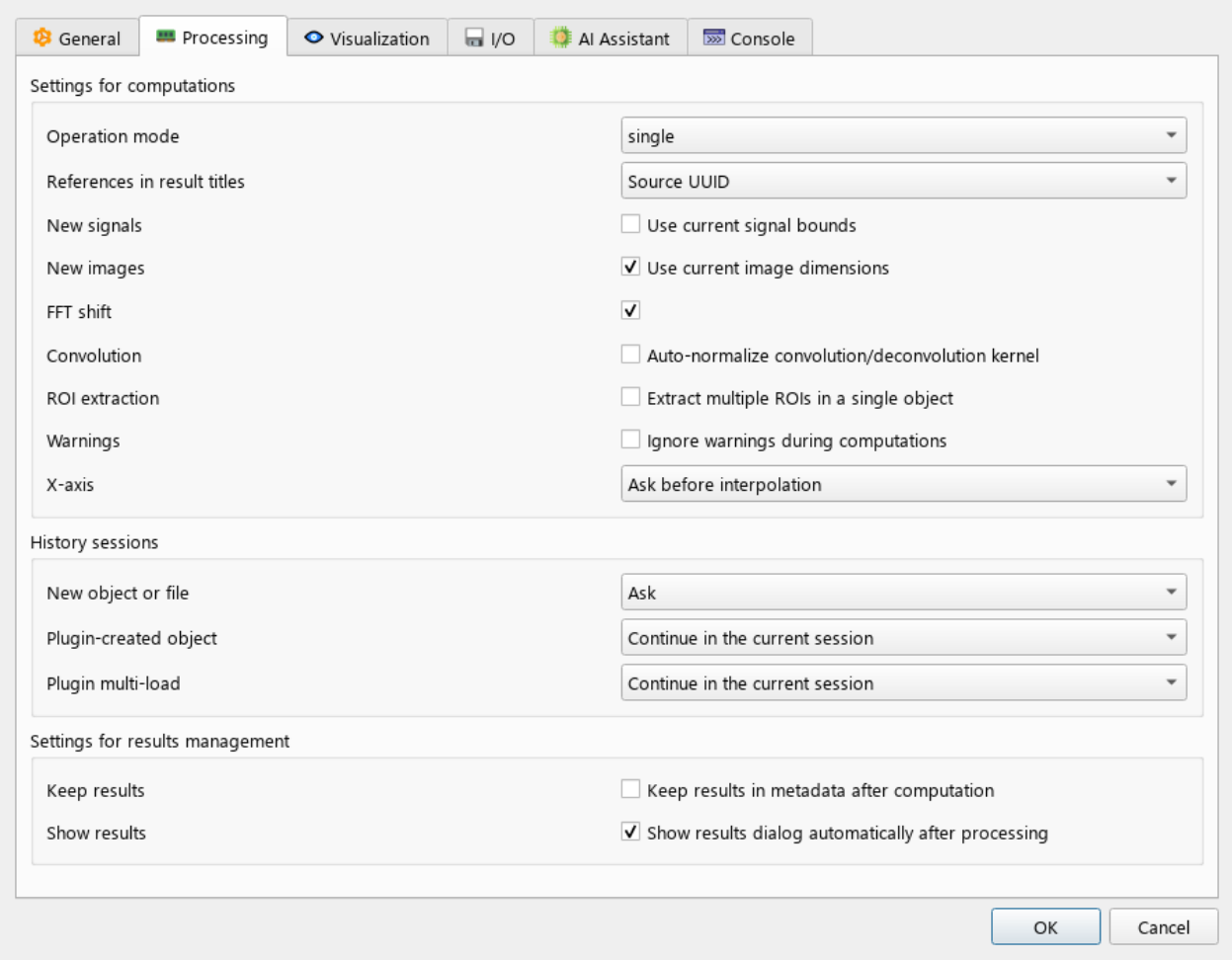


Fig. 4: Processing settings tab

> **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: Σ(A,B) and Σ(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

References in result titles
: Choose how source objects are represented in titles generated for computation results:

  - Source UUID (default): use stable 8-character references derived from each source object's UUID.
  - Source title: use the source objects' live current titles, so references follow later renames. References nested in processing chains are resolved too, e.g. `derivative(derivative(My signal))`.

  This setting changes only the displayed reference, in the object trees, plot legends, plot titles and labels added with Add object title to plot. The title edited in the Properties panel keeps its stored references. Result titles store 8-character references (`g`-prefixed for groups), while each object's identity remains its full UUID. Signal, image and group items in the trees show their own `#UUID8` (or `#gUUID8`) reference below their title.

Use current signal bounds
: 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 current image dimensions
: 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.

Auto-normalize convolution/deconvolution kernel
: When enabled, convolution and deconvolution kernels are automatically normalized, for signals and images, to preserve signal energy and image intensity. This option is disabled by default.

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

Ignore warnings during computations
: Suppress warning messages during computations.

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

  - Ask before interpolation: display a confirmation dialog (default)
  - Interpolate systematically: automatically interpolate signals

History sessions

These settings control how new inputs are assigned to history sessions. They are evaluated only when Record mode is enabled and the shared active session contains actions. No policy decision is needed when the active session is empty; see History Panel for the complete workflow.

New object or file
: Choose what happens when a new object is created or a file is loaded:

  - Ask (default): ask whether to start a new session
  - Always start a new session: start a session before recording the input
  - Continue in the current session: append the input to the active session

Plugin-created object
: Choose what happens when a plugin adds one object:

  - Ask: ask whether to start a new session
  - Always start a new session: start a session before recording the object
  - Continue in the current session (default): append the object without a modal prompt, so plugin execution is not blocked

Plugin multi-load
: Choose what happens when a plugin explicitly groups several object additions in one multi-load scope:

  - Ask once: ask once whether the whole batch should start a new session
  - Start a new session: start one session for the batch
  - Continue in the current session (default): append the whole batch to the active session

  Ask once is the UI label for one durable session decision covering the complete explicit plugin multi-load scope, rather than one prompt per object. With ordinary Ask behavior, repeated prompts for synchronous additions to the same panel are debounced during the current Qt event-loop turn.

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 "Show results…" entry of the "Analysis" menu.

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:

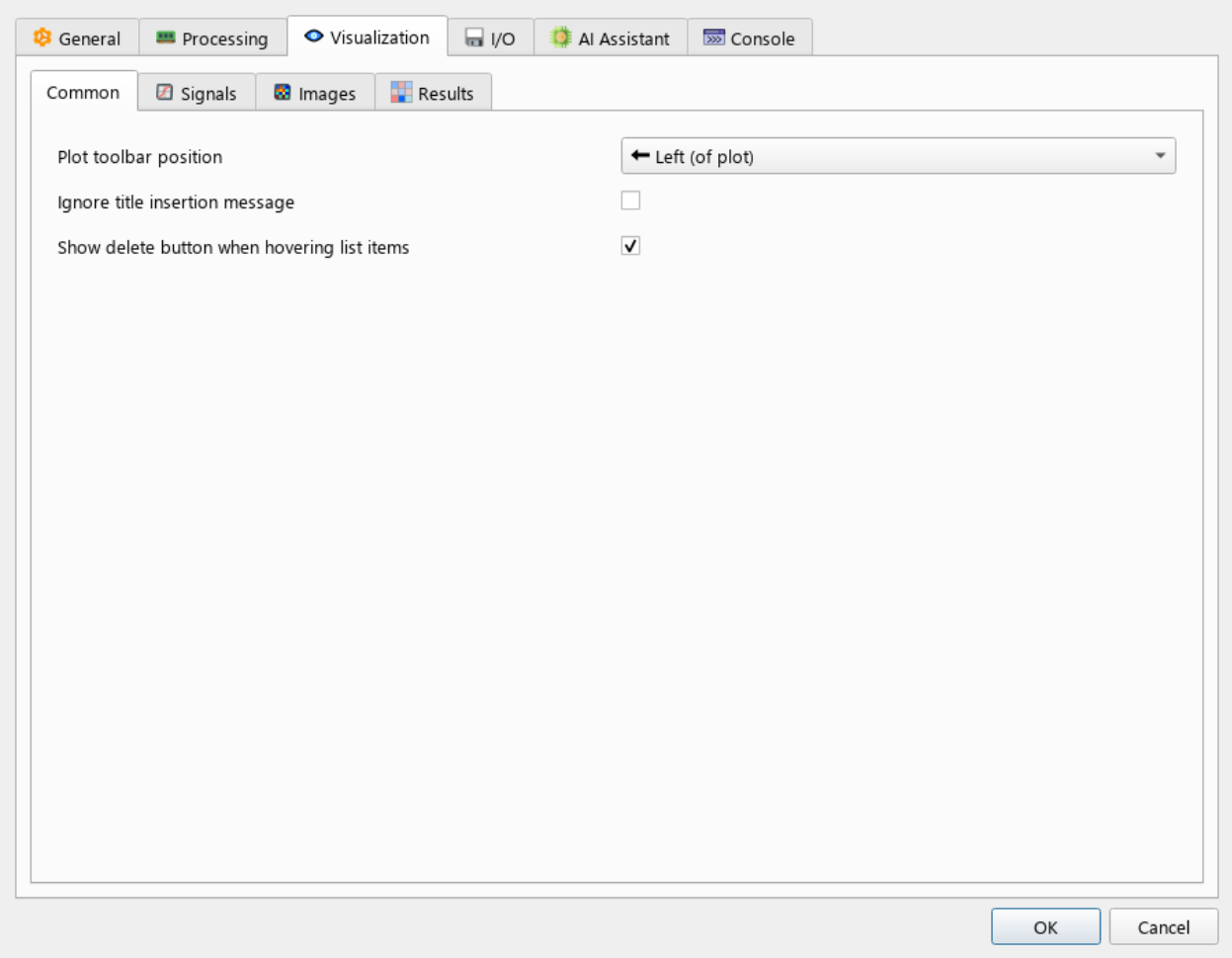


Fig. 5: Visualization settings - Common sub-tab

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.

Show delete button when hovering list items
: When enabled (default), a delete button appears at the right of the signal, image or group under the mouse cursor

in the signal and image lists (see Overview). Disable this option to hide the button: items can still be deleted with the "Remove" action of the "Edit" menu, or with the `Delete` key.

Signals

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

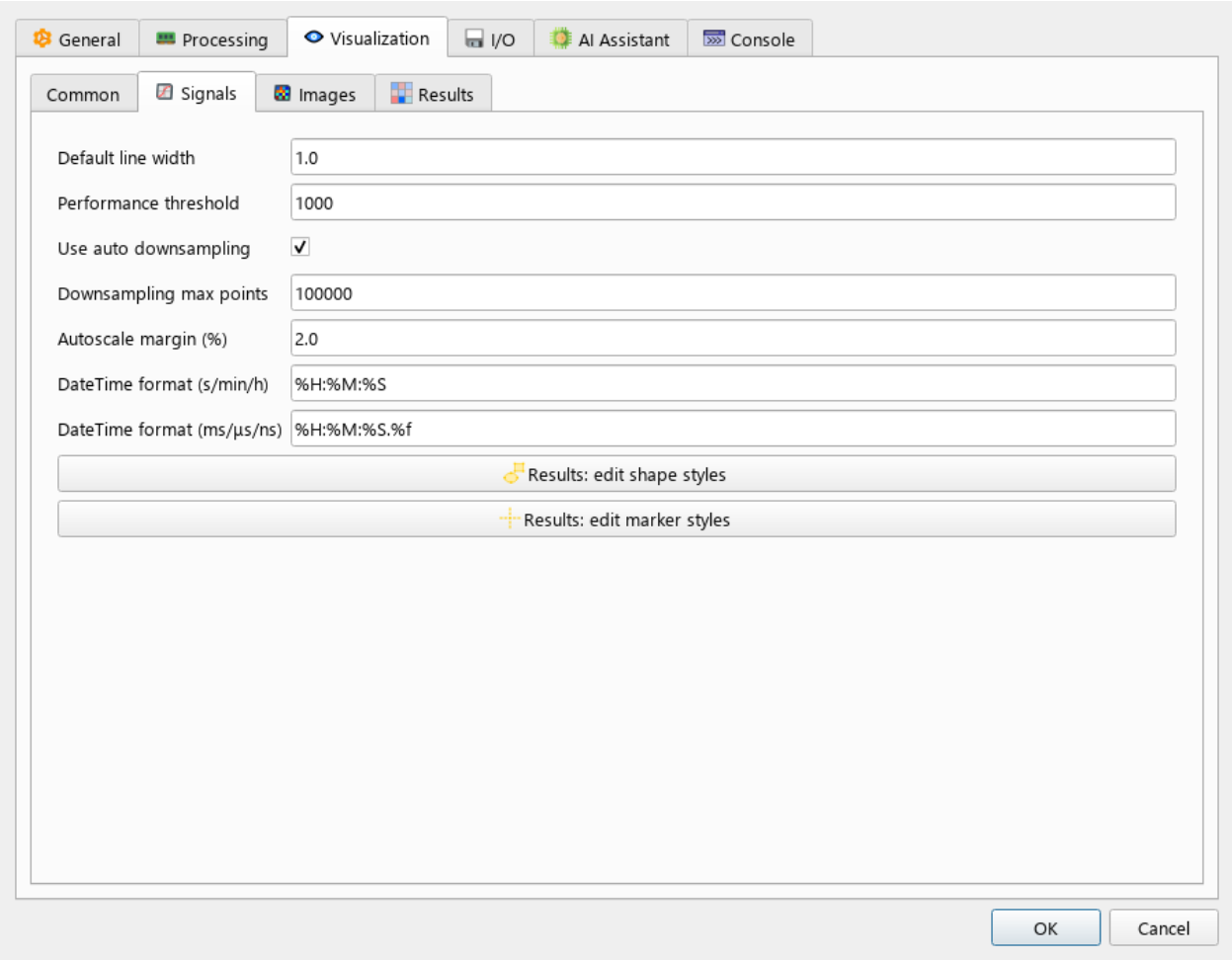


Fig. 6: Visualization settings - Signals sub-tab

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. Curves of width 1.0 or less are drawn one screen pixel wide, also on high-DPI screens (e.g. with a 150% display scaling), which keeps their rendering fast.

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 only affects the display: signal data and processing are not modified.

Downsampling max points
: Maximum number of points displayed for a signal when auto downsampling is enabled (default: 100,000). Signals with more points are downsampled so that at most this number of points is displayed.

Autoscale margin
: Percentage of margin to add around data when auto-scaling signal plots (default: 2%). 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/µs/ns)
: Format string for datetime X-axis labels when using sub-second time units (ms, µs, 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 (default: 1%). 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.).

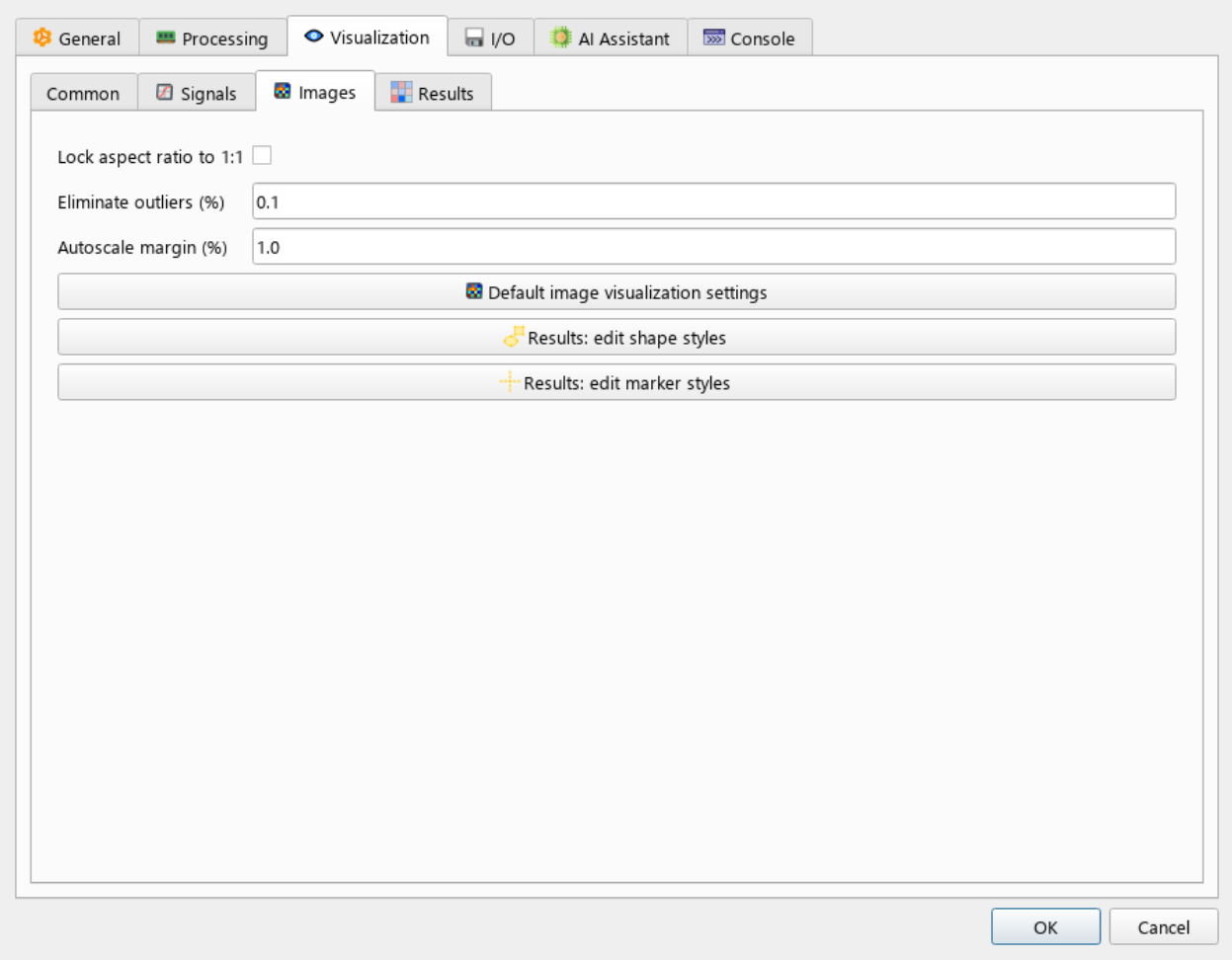


Fig. 7: Visualization settings - Images sub-tab

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.

Results

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

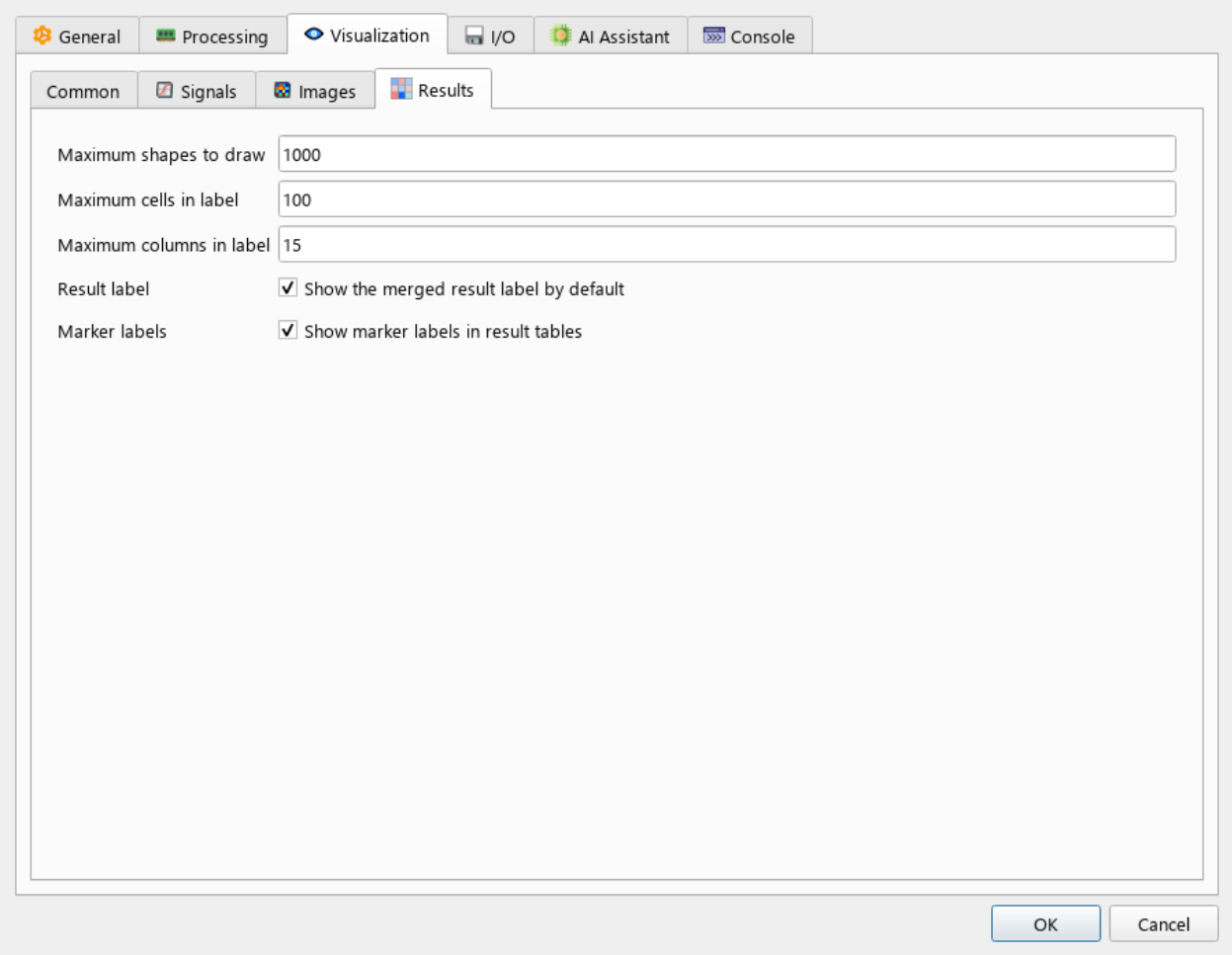


Fig. 8: Visualization settings - Results sub-tab

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.

Show marker labels in result tables
: When enabled, a marker-label column is prepended to result tables for marker results (XY, X or Y markers), so that each row can be matched with the corresponding cross or dashed cursor drawn on the plot. XY markers use numeric labels (`#1`, `#2`, …), axis markers use letters (`a`, `b`, `c`, …).

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

I/O

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

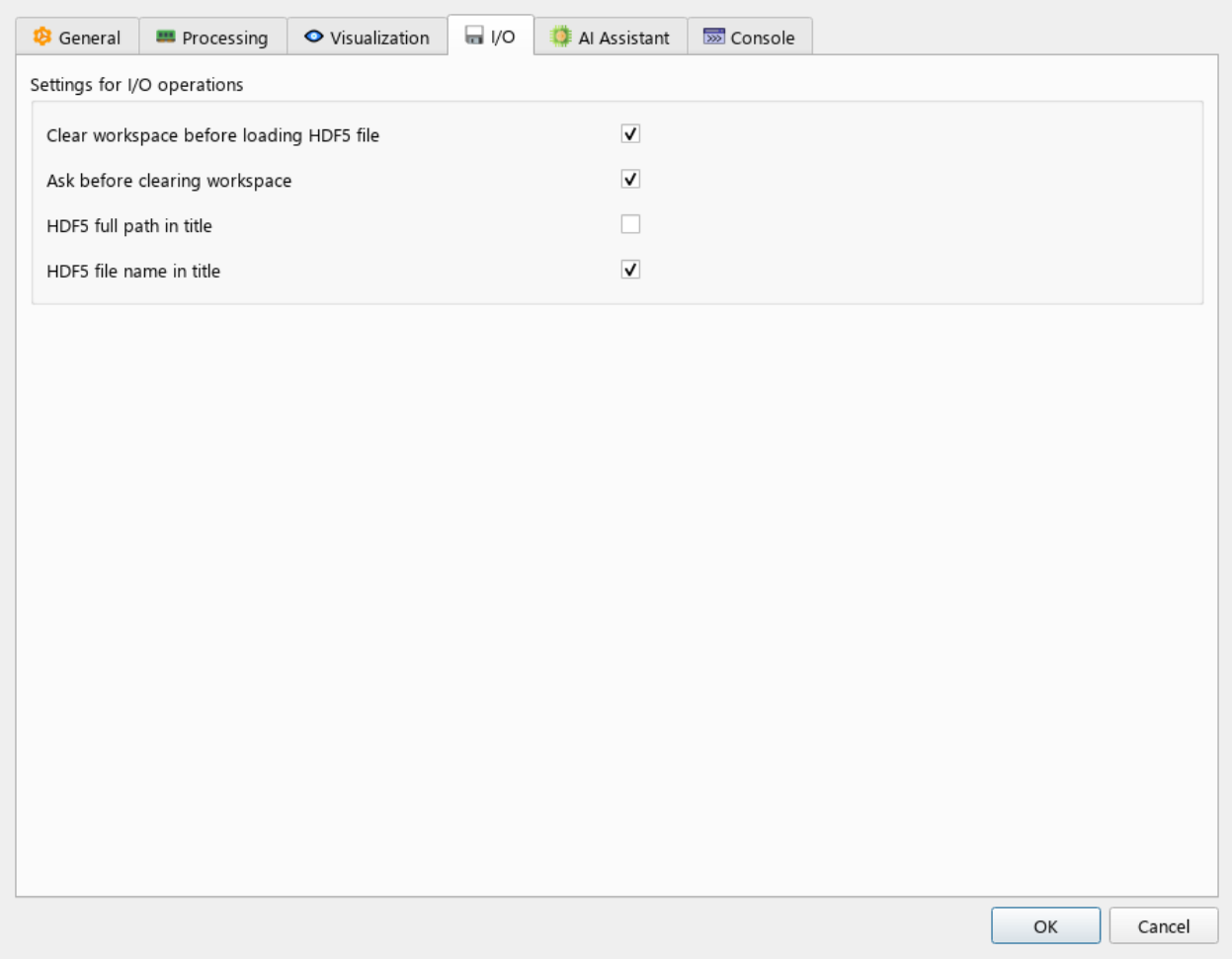


Fig. 9: I/O settings tab

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.

AI Assistant

The AI Assistant settings tab configures the built-in conversational assistant (see AI Assistant). The Settings… button of the AI Assistant panel opens this tab directly. The assistant contacts its provider only when you send a message or click Test connection.

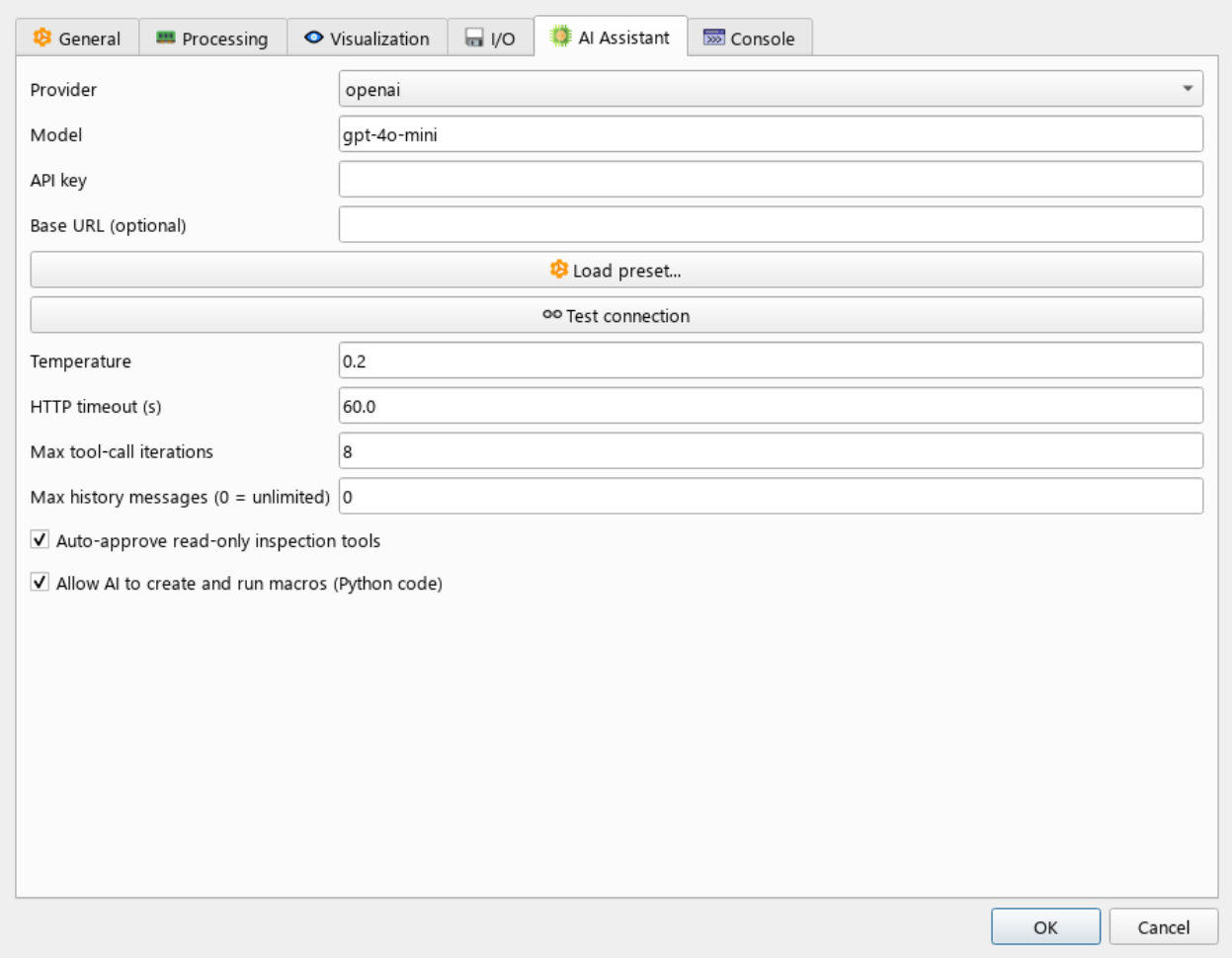


Fig. 10: The AI Assistant settings tab.

Provider
: Backend used to serve the model. `openai` covers every OpenAI-compatible endpoint (OpenAI, GitHub Models, Azure OpenAI, Ollama, LM Studio, llama.cpp, vLLM, …). Use `mock` to try the assistant offline, without any API key: it replies with scripted answers triggered by simple keywords.

Model
: Name of the model to use, as expected by the provider (e.g. `gpt-4o-mini`).

API key
: Provider API key.

> **Warning**
>
> This value is stored in plain text in the DataLab configuration file. It is recommended to leave the field empty and to set the provider's standard environment variable instead (e.g. `OPENAI_API_KEY`): the secret is then never written to disk and can be shared with other tools.

Base URL (optional)
: Endpoint override for OpenAI-compatible services. Leave empty to use the provider default.

Load preset…
: Pre-fill Base URL and Model with a known OpenAI-compatible endpoint (OpenAI cloud, Ollama, LM Studio, llama.cpp, vLLM).

Test connection
: Probe the configured endpoint to check the URL, the API key and network reachability. This does not consume any token.

Temperature
: Sampling temperature passed to the model (0 = deterministic).

HTTP timeout (s)
: Maximum duration of a single request to the provider.

Max tool-call iterations
: Safety cap on the number of tool calls the assistant may chain to answer a single question.

Max history messages (0 = unlimited)
: Cap the number of past messages sent to the provider on each request. Useful with local models having a small context window, where an over-long conversation is rejected by the server. The system prompt and the current user message are always preserved.

Auto-approve read-only inspection tools
: When enabled, tools that only read the workspace (listing objects, inspecting an object, listing available operations) run without asking for a confirmation. Tools that modify the workspace always require an explicit confirmation.

Allow AI to create and run macros (Python code)
: When enabled, the assistant may propose Python macros, executed with full access to DataLab through the proxy API — each execution still requires an explicit confirmation. Disable this option to hide the macro tool from the model entirely.

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. Changing this option requires restarting DataLab.

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.

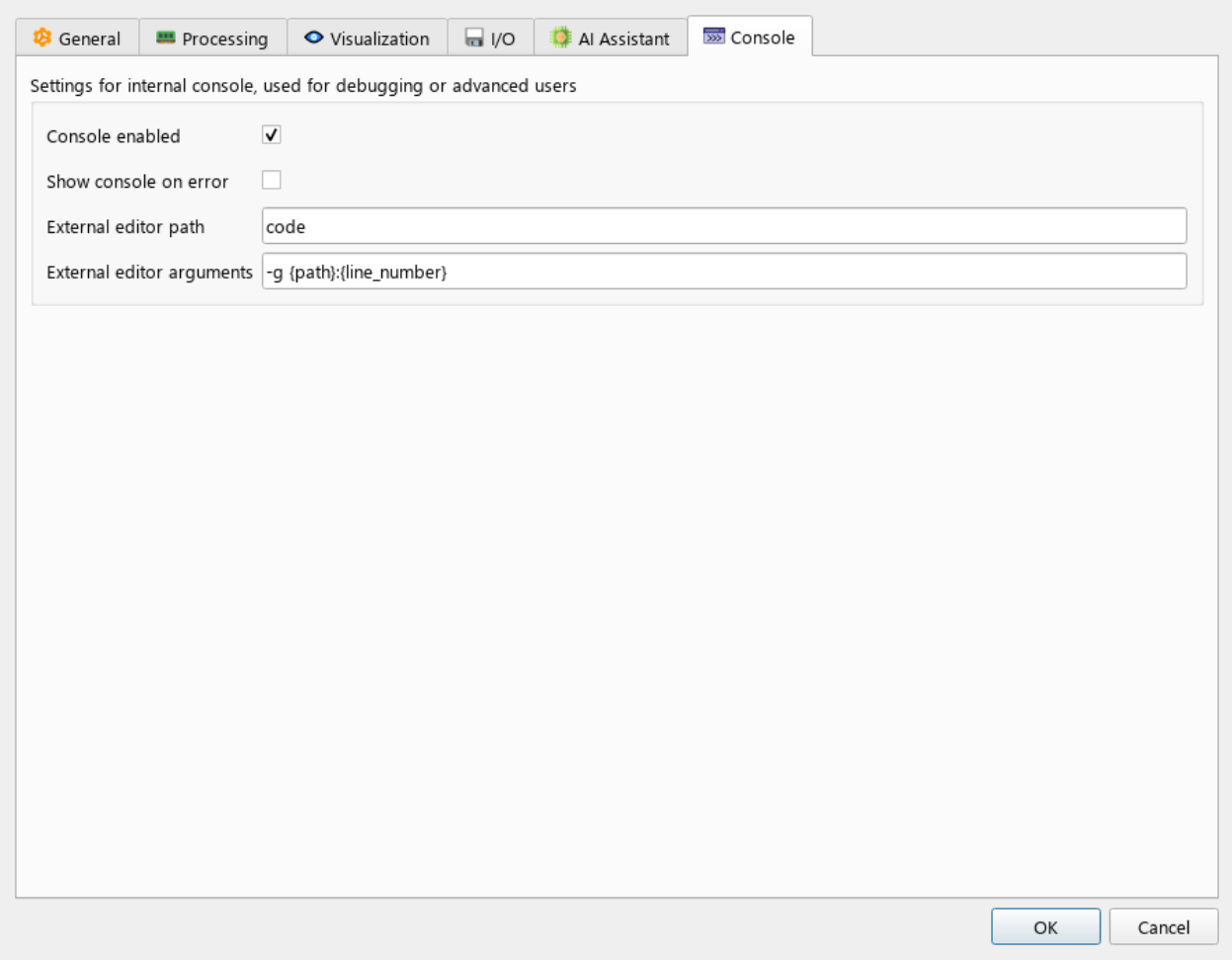


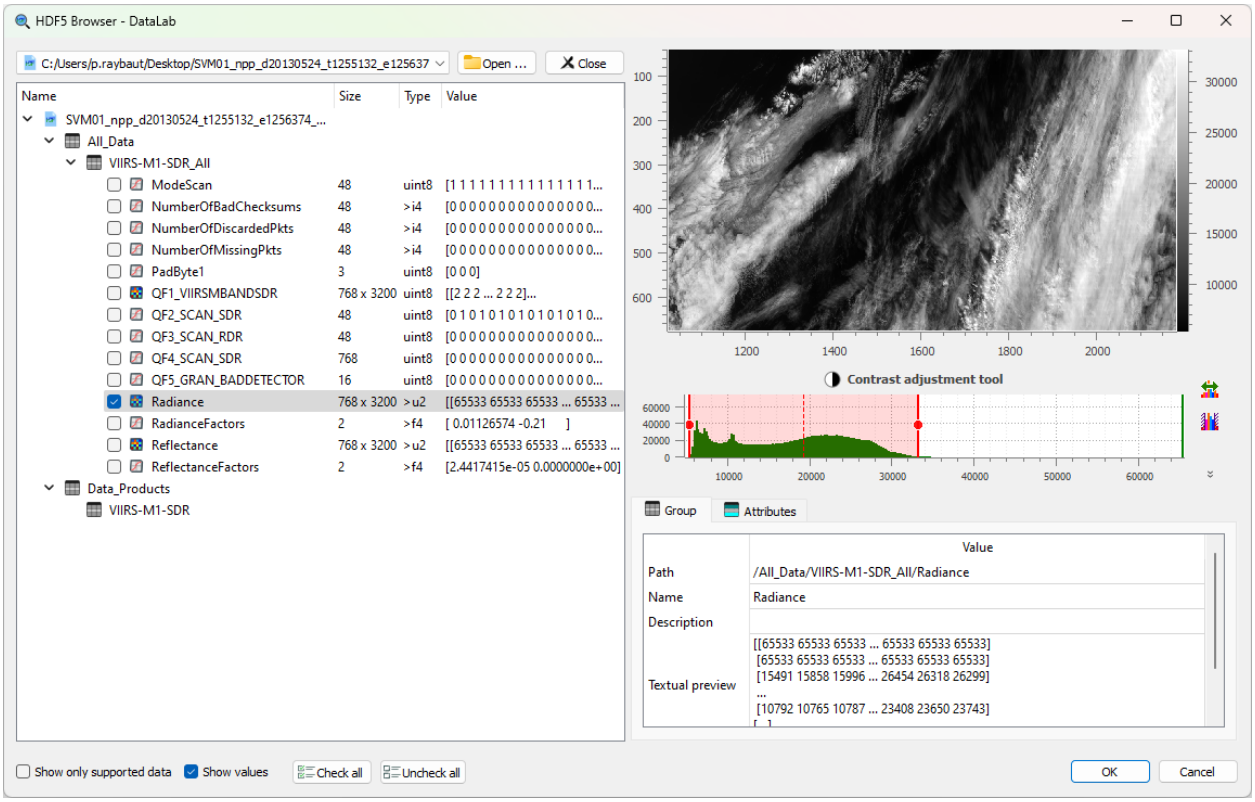
Fig. 11: Console settings tab

External editor path
: Path (or command) of the external editor used to open a source file at the line of an error, from a traceback shown in the console or in a warning or error message box (default: `code`, i.e. Visual Studio Code).

External editor arguments
: Command-line arguments to pass to the external editor: `{path}` and `{line_number}` are replaced by the path of the file and the line number of the error (default: `-g {path}:{line_number}`).

3.1.3 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 (or on "Uncheck all" to clear the selection)
- Then validate by clicking on "OK"

Other commands are available:

- The "Open …" and "Close" buttons, next to the file selector, add another HDF5 file to the browser or close the current one

- The “Show only supported data” check box hides the data that cannot be imported, and the “Show values” check box shows the “Value” column of the tree
- The “Show array” button, at the top right of the right panel, shows the values of the selected array in a read-only table
- The context menu of the tree expands or collapses all items (“Expand all”, “Collapse all”) or the selected ones (“Expand selection”, “Collapse selection”), and “Restore” restores the original tree layout

> **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.

NeXus files and physical axes

Open `.nxs` and `.nx5` files with Browse HDF5 file, just like `.h5` files. A NeXus `NXdata` group designates a measured dataset and its axes. Select the group to import the complete scientific object, rather than selecting its individual datasets: a 1D group produces a signal, and a 2D group an image. The raw datasets remain available separately in the tree.

The import preserves axis values, labels and units, including non-uniform image coordinates. Signal Y uncertainties are imported when supplied; image uncertainties are not imported. Histogram axes supplied as bin edges are converted to bin centers, and this conversion is recorded.

An axis with an invalid shape or length, an `axes` attribute that does not give one axis per dimension, or an axis that does not hold numbers is reported in the group description and prevents importing the group. An undeclared axis uses point or pixel indices; the importer does not infer a physical axis from neighbouring datasets. Images need increasing coordinates: a decreasing image axis is reversed together with the image, and this conversion is recorded, while an image axis that is not strictly monotonic prevents the import.

Files written with the legacy NeXus conventions, such as the scans of synchrotron SOLEIL, are imported with their axes too: an axis dataset may give the dimension it describes with its `axis` attribute, the one with the lowest `primary` attribute being preferred. The other measured datasets of a group, such as all the counters and detectors of a scan, appear as additional items below the group, each with the axes that fit its dimensions: a counter is a signal versus the scanned motor, a detector writing a spectrum at each point an image with the motor along Y, and a detector writing an image at each point a stack of images (see below).

When a NeXus file designates a default measurement (`default` attributes), the browser selects and previews it when the file is opened. Compressed detector data, such as the bitshuffle/LZ4 data of Eiger detectors, is read; a dataset compressed with an unavailable filter is reported in the tree instead of failing.

Files with several counters or incomplete NeXus descriptions can be interpreted by a plugin-provided data mapping. Each mapped object appears as an additional item with its own axes, labels and units; the axes of a mapped object may describe its first dimensions only, so that one mapping applies to the counters and detectors of a scan. Mappings must be registered before opening the file; DataLab does not automatically discover JSON files beside it. Plugin authors can follow the SigimaX HDF5 import guide.

Large datasets and previews

Listing numeric datasets uses their shape and type without loading all their values. Textual value previews read only the first elements, and attributes containing more than 10 000 values are not copied to imported object metadata. Graphical previews are read in a background thread; only the preview of the last selected item is displayed, and items selected meanwhile are not read. Selecting an item that cannot be previewed, such as a group, clears the preview. Broken links, such as external links to missing detector files, are skipped when the file is opened.

Large ordinary 1D arrays and 2D images, NeXus and mapped objects are previewed with a sampling step above `2**24` elements (about 16.8 million). Import still reads every value, not the sampled preview. This is not a universal memory

limit: compound datasets and signals stored as 2D tables still use complete objects for their graphical previews. Closing a file or the browser waits for its preview read to end.

Multidimensional datasets

Datasets with three or more dimensions, such as image stacks and spectral cubes, are imported as one image or signal per frame. Selecting such an item shows a view selector below the preview:

- View: images of two axes, or signals along one axis
- Frames along: the axis along which frames are taken, with the First and Last frame and a Step
- Fixed indices: the index of the other axes, if any
- Import these frames: makes the item checkable and checked; unchecking the item cancels the import

The first frame of the view is previewed, and the number of frames is shown. Images of the last two axes, with every frame along the preceding axis, are proposed by default, unless the NeXus `interpretation` attribute of the dataset designates spectra (signals along the last axis) or RGB images (one image per channel). DataLab asks for confirmation before importing more than 100 frames. The frames of NeXus datasets keep the axes of their displayed dimensions; their title gives the frame indices and, if the other axes have coordinates, the position of the frame (e.g. `frames[3, :, :] (theta=0.3)`). Developers can also read views without creating objects, using the selection API described in the SigimaX guide.

Numerical values and experimental origin

Unsigned 32-bit detector counts above 2 147 483 647 are imported as `float64` images without clipping or rounding. Other integer arrays are also converted to an exact supported type when possible. If conversion to floating point cannot preserve every integer (in particular beyond `+/-2**53` for `float64`), Sigima emits a warning instead of silently promising exact values. Complex signals keep their imaginary part, including single-precision ones.

NeXus and mapped objects carry an `experimental_source` metadata entry describing the source file, group, measured dataset, axes (`"."` for a dimension without axis) and recorded conversions, with the title, start and end times of the NeXus entry, and, for frames, the frame indices and position. Mapped objects also identify their mapping. This entry describes where the measurement came from; it is not the processing provenance shown in the Properties panel and does not make the experiment reproducible.

Save the imported objects in a DataLab workspace to keep both their data and this metadata. Reopening that workspace does not require the original NeXus file or the mapping plugin. Generic raw dataset imports do not automatically receive this structured experimental source.

3.1.4 History Panel

Overview

The “History Panel” records the sequence of actions performed by the user on signals and images, organized into sessions. Each session is a chronological list of either:

- UI actions (creating a new signal, removing selected objects, saving the workspace to HDF5, …),
- computations (FFT, average, Gaussian fit, …) dispatched by the DataLab processors to Sigima, or
- mutations (in-place modifications of existing objects, such as editing regions of interest). See Object mutations.

A recorded session can be:

- Replayed silently or step by step. Replaying recomputes each computation action in place with its recorded parameters: existing output objects are updated, and deleted outputs whose action is still part of the history are re-created under their original identifiers, keeping the downstream processing chain valid. In step-by-step mode, parameters may be reviewed and edited before each step is recomputed. Recorded UI actions are invoked through their own methods and may reproduce their side effects, including creating, importing, or duplicating objects;

- Duplicated as independent processing chains in new history sessions, with the required signal/image objects cloned as part of the operation;
- Saved to a standalone history file (`.dlhist`) or embedded in the workspace when saving to HDF5, so that the full processing chain travels with the data.

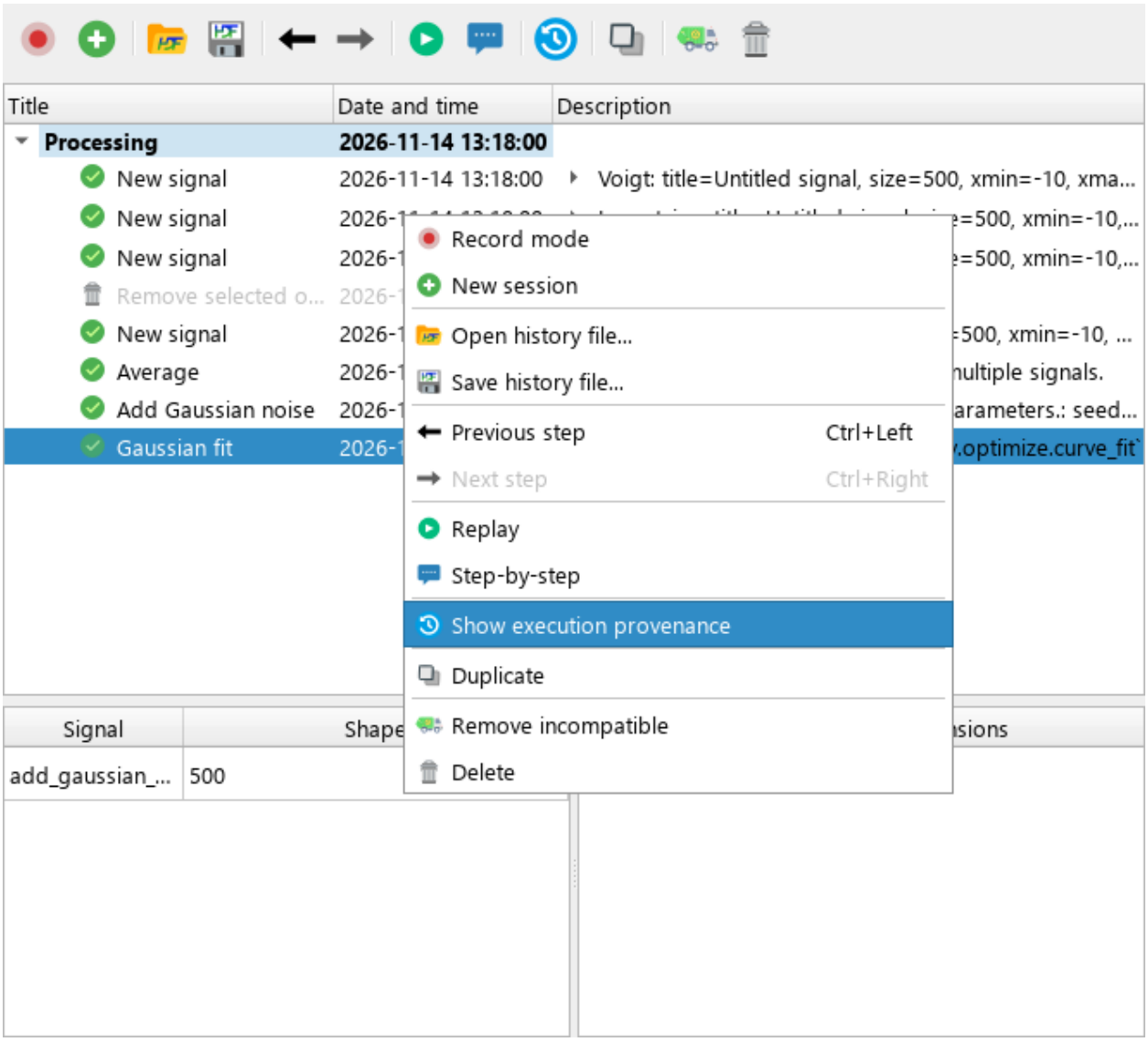


Fig. 12: The History Panel after recording a representative session: create three signals (Voigt, Lorentzian, Lorentzian), remove one of them, create a Gaussian signal, compute the average, add Gaussian noise to the result and run a Gaussian fit.

Object mutations

Besides UI actions and computations, the panel records mutations: in-place modifications of existing objects that do not create new ones. Mutations currently cover regions of interest (ROI): defining or editing ROIs graphically or numerically, deleting one or all ROIs, and pasting ROIs each record a single generic mutation entry holding the affected objects and the resulting ROI state.

Replaying a mutation re-applies the recorded ROI state to its target objects (an empty state removes the ROIs). In step-by-step mode, the ROI parameters can be reviewed and edited in a dialog before being re-applied; editing them triggers

a recompute of the downstream dependent computations.

When a recomputed action re-creates or updates an object, the mutations recorded on that object are re-applied in order, and analyses depending on the mutated object are recomputed. User ROIs present on an output object are preserved by in-place recomputes, unless the recompute itself produces a ROI.

Mutation entries are saved with sessions (standalone `.dlhist` files and HDF5 workspaces) like any other action; history files created with earlier versions of DataLab load unchanged.

Recording and session lifecycle

Actions are recorded only while Record mode is enabled. By default, this mode is disabled when DataLab starts: nothing is recorded until it is enabled. Turning record mode off preserves existing sessions but does not add new entries.

There is a single active recording session, shared by the Signals and Images panels. New actions from both panels are chained into that session (one is created on first use), so mixed signal/image pipelines are recorded together and recording resumes in the user-selected session. Each session remains associated with the Signals or Images panel for display purposes, through the actions it contains.

When a new object is created or a file is loaded into a populated active session, a configurable policy determines whether DataLab asks, starts a new session, or continues the current one. Plugin-created objects use separate policies. An explicit plugin multi-load scope supplies one durable session policy for the whole batch. With ordinary Ask behavior, repeated prompts for synchronous additions to the same panel are debounced during the current Qt event-loop turn.

These options are available under `File > Settings > Processing > History sessions`. See History sessions for the complete labels and default values.

Toolbar

The toolbar at the top of the panel exposes the following actions:

- Record mode: toggle the recording of new actions. When off, no new entry is added to the history (existing sessions are preserved).
- New session: start a new history session and make it the active recording session.
- Open history file: load recorded sessions from a standalone `.dlhist` file.
- Save history file: save the current recorded sessions to a standalone `.dlhist` file.
- Previous step: select the preceding action in the current session (keyboard shortcut: `Ctrl+Left`).
- Next step: select the following action in the current session (keyboard shortcut: `Ctrl+Right`).
- Replay: recompute the selection in place, silently (no parameter dialogs). Selecting an action replays that action; selecting a session replays all of its actions. A selection spanning several actions or sessions is merged, deduplicated and executed in session order. Each computation action re-runs with its recorded parameters and updates its existing output object(s), keeping the same identifiers so that downstream steps remain valid. Outputs that were deleted from the data panel are re-created under their original identifiers (a typical workflow: delete a bad result, edit its parameters, then replay to regenerate it). Actions whose source objects no longer exist are skipped with a warning, and a failed action blocks its downstream branch. Actions whose parameters were changed in step-by-step mode are marked as outdated; replaying recomputes them and, when parameters were edited, their downstream dependent actions as well. Analysis actions replay by recomputing their results on the source objects: each analysis records which results it stored on the object (its effects), so replaying updates exactly those results — previous values are replaced, and if the recompute fails the previous results are restored. Analyses recorded

with earlier versions of DataLab replay using their saved state. UI actions are replayed by invoking their recorded method and may reproduce side effects, including creating, importing, or duplicating objects; load actions reload their objects even if these were deleted after being recorded. Recorded destructive actions (e.g. object removals) are skipped with a warning whenever replaying them would be unsafe: when their captured state no longer matches the current workspace, when their targets were re-created earlier in the same replay, or when their targets belong to another session. As a consequence, a replayed session may intentionally not reproduce a recorded end state that included such deletions.

-  Step-by-step: replay the same selection one step at a time, opening the parameter dialog for each supported action (object creation, computation, ROI extraction) before recomputing it. Accepted edits propagate to the downstream dependent actions, which are recomputed as well. Cancelling a dialog stops the replay, restores the parameter edits made during that run, and silently recomputes any actions left outdated so the chain stays up to date.
-  Show execution provenance: show the recorded provenance of the last successful execution of the selected computation action (one tab per result), see Provenance of results. If the parameters of the action were edited without executing it again, the dialog says so.
-  Duplicate: duplicate the processing chain containing each selected action, or the processing chains in each selected session. DataLab clones the required objects and creates independent history sessions. The duplicate is a standalone deep copy of the chain — as if the recorded operations had been redone by hand on the cloned objects — so replaying either the original or the duplicated session only affects its own objects.
-  Remove incompatible: remove all actions whose workspace state is no longer compatible with the current workspace. A confirmation dialog shows how many actions will be removed.
-  Delete: remove the selected actions or sessions from the history (keyboard shortcut: `Del`, active only while the history tree has focus, so it never deletes signal or image objects instead). Removing an intermediate action splices it out and preserves its downstream steps as an independent chain.

> **Note**
>
> Double-clicking a tree item invokes Replay for the current selection, with the same in-place recompute semantics documented above.

> **Note**
>
> While a replay or recompute is in progress, History Panel commands are unavailable: the action toolbar commands are disabled, and tree interactions (context menu, double-click, `Del` key) are ignored until the run completes. The record-mode control remains available.

Tree view

The tree view organizes recorded actions into expandable sessions:

- Each top-level row is a session; the panel(s) it relates to are determined by the actions it contains. Sessions may be started when recording is enabled, with New session, or according to the configured session policy.
- Each child row is an action, with its title, date/time and a description summarising its parameters or resolved call when available. A UI action whose call cannot be resolved may have an empty description.

The selection of one or several rows determines which entries are targeted by the toolbar and context-menu commands. The context menu exposes the same commands as the toolbar.

When an action row is selected, its result object is selected in the corresponding data panel when available; otherwise, its existing input objects are selected. DataLab then switches to that data panel.

While Record mode is enabled, selecting a session row (or one of its actions) makes that session the active recording session.

Actions that are not compatible with the current workspace state (for example because a referenced object identifier no longer exists, or because its data array shape changed) are shown with a disabled foreground and an explanatory tooltip. Attempting to replay them opens the broken chain resolution dialog described below.

Actions left outdated by a failed or interrupted recompute, or by parameter edits whose propagation to downstream steps is still pending, are highlighted with an amber background and carry an explanatory tooltip. Replaying them refreshes their result and clears the marker.

Broken chain resolution

Starting Replay or Step-by-step (or double-clicking a tree item) on a selection containing actions that are incompatible with the current workspace — typically because the objects they rely on were deleted — opens a resolution dialog. Two choices are offered:

- Repair and continue: the broken actions and all their downstream descendants are removed from the history, then the remaining valid actions of the selection are replayed.
- Cancel: nothing is modified, neither the history nor the workspace. The dialog is shown again on the next replay attempt until the chain is repaired.

Workspace state display

Below the action tree, a split-view widget shows the workspace state captured at the time of the selected action:

- Left table: lists the signals that were selected, with their array shape.
- Right table: lists the images that were selected, with their dimensions.

This information helps the user understand the context in which each action was originally executed and diagnose compatibility issues when replaying the current selection.

Persistence

The history can be persisted in two complementary ways:

- Embedded in the workspace: when the workspace is saved to HDF5 (`File > Save to HDF5 file`), the History Panel content is automatically saved alongside the signals and images. Reloading the workspace restores the recorded sessions.
- Standalone history file (`.dlhist`): the file embeds both the recorded sessions and all objects currently present in both the Signals and Images panels, whether or not an action references them. This makes the file fully self-contained:
  - Opening a `.dlhist` into a pristine workspace (with no data objects and no existing history sessions) restores the saved objects and sessions directly.
  - If the workspace is already in use (it contains any data object or history session), DataLab imports the objects into new signal/image groups, remaps their identifiers to avoid collisions, and appends imported history sessions that reference those fresh identifiers.

> **Note**
>
> A history entry whose saved parameters can no longer be decoded (for example a ROI payload written by an incompatible version of DataLab) does not prevent the rest of the file from loading: the affected action is loaded as incompatible — shown as a disabled row with an explanatory tooltip, and never replayed — while all other sessions and actions load normally. Such entries can be cleaned up with Remove incompatible.

> **Note**
>
> Each action keeps, in the optional `provenance_activity_ids` field (History action schema version 3), the identifiers of the provenance activities recorded by its last successful execution. This field is never copied with an action (duplicated chains and imported sessions have none), and actions saved by earlier versions simply have none. Deleting a session does not change the provenance record of the workspace.

> **Warning**
>
> Replaying a session that depends on external files (e.g. opening a dataset from disk) will only succeed if those files are still available at the same locations as when the session was recorded.

Chain reconnection on deletion

When a result object is deleted from the signal or image panel (not from the History Panel tree), and that object was produced by a recorded processing step, the History Panel automatically reconnects the processing chain:

- All downstream steps that consumed the deleted object are rewired to use the source of the deleted step as their new input.
- For `2_to_1` operations (e.g. difference), the first source is used for reconnection.
- If the source of the deleted step no longer exists (e.g. it was already deleted, or is deleted at the same time), the deletion proceeds and the dependent results are left unchanged: a single warning lists the processing steps concerned, each of them once.
- Deleting an object that was not produced by a recorded processing step (e.g. a created or loaded object, or any object when recording was off) leaves the results depending on it unchanged, without any warning.

This behaviour mirrors removing a link from a chain: the adjacent links reconnect to preserve the processing flow.

> **Note**
>
> Reconnection is only triggered by deletions initiated from the signal/image panels. Deleting an action directly from the History Panel tree behaves differently: the selected action is spliced out instead of truncating the session. If downstream steps depend on it, DataLab preserves them as an independent chain by cloning the required intermediate object and reconnecting those steps to the clone. Deleting a session removes that complete session.

Auto-recompute

> **Note**
>
> When a result object is selected in the signal/image panel and it has editable processing parameters, a Processing tab appears in the Properties panel. Checking Auto-recompute on edit in that tab will re-run the computation automatically 300 ms after any parameter modification.

3.1.5 Provenance of results

For supported processing paths, DataLab records how a result was obtained: the processing or analysis applied, its parameters, the objects it used with their exact content, and the software versions. This record is saved with the HDF5 workspace and read back when the workspace is reopened, by DataLab or by DataLab-Web.

This page describes how to consult that record for one object, how to check and verify it, how to apply a recorded processing again, and how it relates to the History panel. To exchange a workspace with the provenance of its processing, see Workspace capsules.

Consulting the provenance of an object

Select an object and open the Provenance tab of the Properties panel. When the workspace holds a provenance record for this object, the tab shows its provenance:

- a summary: the recorded execution that produced the current content, whether the data still have to be checked, and any limitation (e.g. data altered in the file, or a state observed without recorded production);
- the Production of the object, its Recorded analyses, its Earlier recorded states and, for workspaces saved by older versions, the Records of an old workspace and the Metadata history (not verified).

Each recorded step is shown as a card: the operation, how and when it was recorded and by which software (DataLab Desktop or DataLab Web), its parameters, its inputs and results with their status, and links acting on what they are next to. Links marked with a triangle (▯) show a hidden part of the view and become ▯ to hide it again; an earlier step is shown framed inside the card of the step that used it:

- Show recorded step (under an input) shows how that input was itself produced, step by step, without computing anything. The state actually used is shown, even if the input object was modified or reprocessed since: for example, if `S2` was computed from `S1` filtered with sigma = 2 and `S1` was then filtered again with sigma = 5, the recorded step of the input of `S2` shows sigma = 2. Hide recorded step hides it again;
- Go to object (under an input or a result) selects, in the signal or image panel, the object that holds it today (it may have changed since the computation);
- Verify this step recomputes the step separately and compares every result with the recorded one: Exact, Within tolerance, Different or Not verified (with the reason, e.g. an original input missing). The workspace is not changed;
- History action selects the History action whose last execution recorded the step, when there is one;
- Show details shows the context of the step (options, X alignment, metadata read) and the software versions.

Consulting reads the provenance record only: it never computes, never changes the workspace and never adds a History action. The view is bounded and long values are abbreviated: a truncated view does not mean that records were removed.

The buttons below the cards act on the object:

- Check data compares the object, the direct inputs and results of its production and the earlier steps shown with their recorded states, without computing. The steps shown and the details opened stay so. A deleted input or result is reported as missing at once, without checking (nothing has to be read to know it);
- Verify computation verifies the production of the object, like Verify this step.

States and messages

The provenance never invents what is unknown. The main messages are:

Message	Meaning
Data to be checked	Nothing was compared yet: use Check data
This input has changed since the computation	The input object no longer holds the state used by the computation; the recorded state is still described
Original input missing	The input object was deleted (or its identifier is now held by another object, see Appending a workspace)
The result was modified since the computation	The result was edited after it was computed: it is reported as altered rather than trusted
The data of the object are not available	The data cannot be read to be compared (they are not checked)
The state cannot be checked (no fingerprint)	The record does not allow a comparison
Observed state, without recorded production	The object exists in the record, but how it was produced is unknown (e.g. data loaded from a file)

Applying a processing again

When the latest recorded state of an object was produced by a processing of DataLab, the Processing tab of the Properties panel restores that call: its operation, its parameters and every input, in order and with its role (source, operand, or each aggregated object). This applies to processing combining several objects (e.g. difference of two signals, average of several images) as well as to 1-to-1 processing, and to workspaces reopened from a file, saved by DataLab or by DataLab-Web.

- With parameters, edit them and click Apply;
- Without parameters, click Reapply.

The result is updated in place with the recorded inputs, never with the current selection, and the new execution is recorded. When the processing cannot be applied again, the tab says why (e.g. an input object no longer exists, the recorded parameters cannot be used, or the operation is not available in this version of DataLab).

When no provenance is recorded for an object, the processing metadata stored with it are used as before.

Link with the History panel

When the History panel records a session, each action keeps the identifiers of the provenance activities recorded by its last successful execution. Select an action and use Show execution provenance (History panel menu or toolbar) to display them, one tab per result (or per analysed object). If the parameters of the action were edited without executing it again, the dialog says that the provenance describes the last successful execution.

The History action is the recipe, the provenance records each execution:

- applying a processing again from the Processing tab, or an analysis from the Analysis tab, is recorded in the provenance of the result, and leaves the History action unchanged: replaying it applies its own parameters again;
- replaying the action with Step-by-step in the History panel lets you edit its parameters: the edited parameters become those of the History action, which is linked to this new execution.

Automatic provenance capture does not depend on the History panel's Record mode. Recording a recipe and recording a scientific execution are different operations.

These identifiers are stored in the optional `provenance_activity_ids` field of History actions (History action schema version 3). They are never copied with an action, and actions saved by earlier versions simply have none. Deleting a History session does not change the provenance record.

Appending a workspace

When a workspace file is opened without clearing the current workspace, its data are added but its earlier provenance is not imported: the provenance record of the current workspace is kept unchanged, and later processing of the added objects is recorded as usual. An added object never inherits the provenance of another object, even when it holds the identifier of an object that was deleted from the current workspace: such an object is neither shown nor used as the original input or result of a recorded step, including when verifying it.

DataLab states this limitation before anything changes:

- when DataLab asks whether to clear the current workspace, the files are chosen first and, if one of them holds provenance, the question says so;
- otherwise, a single warning offers to Add without provenance or to Cancel.

To keep the provenance of a file, open it replacing the current workspace. When workspaces are loaded from a script with *load_h5_workspace()*, no question is asked and the notice is logged.

What the provenance does not do

- It does not keep every earlier array. Earlier states are described by their fingerprints; their data remain available only while an object holds them. Deleting or replacing an input's numerical data can prevent replay even after saving and reopening. For example, saving a filtered signal after deleting its source saves the description of the source, not its arrays. Keep the necessary inputs and the recorded results before saving.
- ROI-only changes are a limited exception. If an input's numerical data are unchanged, its recorded ROI can be restored on a copy for verification, including when a detection added ROI. This does not reconstruct missing values. A missing or altered output is still not a valid comparison reference: restoring an input's ROI does not repair the reference.
- Analysis values are not all archived. A later analysis can replace a result under the same metadata key. The earlier artifact stays described as `replaced`, without necessarily retaining its old value.
- It does not clean up the workspace. Trials and intermediate objects stay until you remove them. Their recorded steps remain in the provenance record: deleting an object never purges its activities or sensitive information.
- It does not make everything verifiable. Changes made outside recorded processing may be observed later without a known production. An operation without a contract can be recorded as opaque, but is never replayed. Old metadata records have unknown input states and possibly incomplete parameters or environments; they are not qualified retroactively or executed from their Python names.
- It is not a proof of scientific validity. Exact or Within tolerance says that the recorded computation is reproduced under the rule declared for that operation, not that the processing was the right one.
- It does not identify the author. Fingerprints show that the content was not altered; they do not authenticate who produced it, and capsules are not signed.

Capture scope and verification conditions

Ordinary and grouped processing, analyses, signal/image conversions and ROI extraction use the supported capture paths. Processing, Recompute and replaying a supported computation from the History panel record a new execution; verification itself works on copies and records no new execution. An accepted preview can become a committed result, but a discarded preview, a failed calculation or a cancellation is not a successful execution. A successful detection finding nothing is a normal recorded result.

Importing or creating an object does not record how its physical acquisition or earlier conversions were performed. Editing data, ROI or metadata directly does not by itself record a scientific call. Interactive Desktop fits and creating a signal from a markers table are currently not recorded scientific calls, even when an editable History action exists.

Plugins, macros, scripts and remote API clients are captured when they use the supported processors. Code that creates or modifies objects directly can bypass them. Capture does not block processing: if it fails, a result may exist without an activity; the failure is logged and counted, not silently repaired.

A replay needs a local operation contract qualified for the recorded version, decodable parameters, applicable context, available input data and valid output references. The signal and image phase spectra are identified but not qualified. Unknown operations, versions, codecs or policies produce an explicit refusal: the reader never downloads code to guess their meaning.

Sigima increases the version of an operation contract whenever a change of the computation changes its result, whatever the version of DataLab. Activities recorded with an earlier version of the contract are then refused explicitly (operation version not supported) instead of being replayed with a different result, and their processing cannot be applied again as recorded: compute the result again to record it with the current version. The same applies to the versioned rules of the execution context, such as the selection of image pixels by regions of interest. See Operation contracts and versions in the Sigima documentation.

Scientific fingerprints cover numerical data, types, coordinates and units; ROI are checked separately. Titles, display settings and other metadata are not automatically covered. Only declared dependencies such as global options, X alignment or metadata read by the computation are reproduced. For example, changing fit parameters needed by `evaluate_fit` can refuse replay even if the signal values are unchanged.

See the common rules and capture and interoperability scope for the conditions and the paths covered. Exchange, privacy and revision limits are described in Workspace capsules.

3.1.6 Workspace capsules

A capsule (`.dlcapsule` file) holds a workspace and a description of the provenance of its processing (see Provenance of results), so that someone else can check it and replay it, with DataLab or with DataLab-Web. Capsules are created, read and checked by DataLab-Capsule: opening or inspecting a capsule never executes anything it contains.

Capsules are handled from File > Workspace capsule…, a dialog with the same tabs as in DataLab-Web. Three results are always shown apart:

- the integrity of a capsule, computed from its files (nothing is recomputed);
- the supplied documents (verification reports and reviews): they are checked against the capsule, but their verdicts are what their authors claim;
- the local replay of an activity of the open workspace, recomputed by DataLab and compared with its stored result.

Export

Exports the current workspace as a capsule:

- Name and Description of the capsule;
- Revision of …: shown when the workspace was opened from a capsule, also after it was saved as a workspace file and opened again. When checked, the new capsule names that capsule as its parent revision. The attachments of the parent are not copied: the dialog lists them, so that they can be selected again if they still apply;
- Reference the workspace instead of including it: the capsule then holds the address (absolute IRI) where the workspace will be published, with its size and SHA-256, and the workspace is saved as a separate HDF5 file, to be published at that address;
- Attachments: light files (protocol, notes, small tables, 16 MiB at most each) declared in the capsule with their size and fingerprint. They are never opened or executed.

Export capsule… asks where to save the capsule (and, for a referenced workspace, the workspace file).

> **Warning**
>
> Review the data, metadata, titles, trials, timestamps and attachments before sharing. Nothing is anonymized or redacted automatically. Deleting an object does not guarantee that its traces disappear from the provenance record.

The export carries the current workspace, its record and the chosen attachments, not every external resource used upstream. It does not automatically select the minimal analysis needed for one result: trials and intermediate objects may travel with it. Copying only a result to another workspace does not make that workspace a complete verifiable analysis. Prepare a reduced exchange by deliberately keeping or rebuilding the useful inputs and checking what remains verifiable, not by editing the record by hand.

The software licence does not license the data. The manifest contains a workspace licence declaration; when none is supplied, it uses the neutral statement "No licence was specified for this workspace."

Inspect

Choose a capsule… checks a capsule without opening it and shows:

- its summary: name, description, fingerprint, parent revision, whether the workspace is included or referenced, licence, and its activities (which ones can be replayed);
- its Attachments, which are checked against the manifest and saved only when you click Save…;
- its Integrity: archive, manifest, workspace and stored states. For a referenced workspace, Select the referenced workspace… supplies the file, which must match the size and SHA-256 declared in the capsule;
- the Supplied documents added with Add a verification report… and Add a review…, each checked against the capsule.

Write a review creates a review of this revision: a separate JSON file (reviewer, statement, optional verdict endorsed, disputed or comment, cited verification reports). The capsule itself is not changed.

Open as workspace replaces the current workspace by the workspace of the capsule (after confirmation when the workspace is not empty). It is available only for an intact capsule. A capsule exported later from this workspace is proposed as a revision of it. Opening a capsule is recorded in the History panel when recording is enabled.

For a referenced capsule, obtain the HDF5 file separately: its address is not downloaded automatically, and its size and fingerprint must match. A registry does not deliver that workspace on your behalf.

Local replay

Lists the recorded activities of the open workspace. Replay recomputes an activity separately and compares every result with the stored one: Exact, Within tolerance, Different or Not verified (with the reason). The workspace is not changed. Save report… saves the verification report as a JSON file, to be supplied with the capsule or cited by a review.

Revisions

Lists the revisions exported, inspected or retrieved from a registry on this computer (with their parent revision), and compares any two of them: scientific, presentation, storage and integrity, environment and attachment changes, with a scientific conclusion that never claims equality beyond what the capsules record. The list is kept in the DataLab configuration folder (100 revisions at most); it is a convenience, never a source of truth: the capsule files are.

A capsule is immutable and names at most one direct parent. If you open A and export B, the next export still proposes A as its parent, not B; saving and reopening the workspace preserves that origin. Inspecting another file does not change it. Adding a workspace keeps the current record and parent without merging provenance (see Appending a workspace).

Two exports can have different fingerprints without a new calculation: HDF5 storage, dates or presentation can change. `none_detected` means no scientific change detected in the recorded activities and states, not equality of everything

outside the record. Changing an attached protocol is listed under attachments; the reader, not the comparison, judges its scientific importance. There is no automatic merge of concurrent work, collaborative editing or conflict resolution.

Registry

A registry (optional, served by DataLab-Capsule) publishes capsule manifests, their attachments and reviews, never workspaces. Everything it returns is checked by DataLab.

- Registry address and Publishing token: the address is saved in the configuration; the token is kept for the current session only;
- List revisions lists the published revisions; Retrieve checks a revision and its reviews, adds it to the Revisions tab and offers its stored attachments, checked against the manifest;
- once a registry is set, the Inspect tab offers Publish to the registry (manifest, attachments and valid supplied reviews) and Publish the review.

The reference registry has no accounts, strong author identity, quotas, rate limiting, delete API or fine-grained permissions. Its publishing token is not proof of scientific authorship. See the existing deployment and trust guidance. Reviews and reports target one revision; a checked author's assertion does not replace local replay or implement a complete approval workflow.

From scripts

Capsules can be exported and opened from scripts and macros (see Remote controlling (XML-RPC)), with *export_capsule()* and *open_capsule()*:

```
fingerprint = proxy.export_capsule(
    "chain.dlcapsule", name="chain", attachments=["protocol.txt"]
)
proxy.open_capsule("chain.dlcapsule")
```

The `workspace-capsule` command line of DataLab-Capsule also validates, inspects, compares, verifies and publishes capsules.

Its `verify` command performs no Sigima calculation. Without h5py, workspace checking is limited to size and SHA-256 (`integrity_only`), not individual HDF5 states. A referenced workspace not supplied further limits the checks.

Limits

DataLab checks the size of a capsule, and of the workspace it references, before reading it, and never writes a capsule it would refuse to open. These limits are those of DataLab-Capsule (a workspace of 4 GiB at most): they are accepted sizes, not a tested capacity.

What was measured, on a laptop of 2026 (32 GiB of RAM):

- inspecting or opening a 32 MiB capsule from the dialog takes 0.2 to 0.5 s, during which DataLab does not respond (wait cursor); exporting it takes 0.25 s. Larger capsules take longer, roughly in proportion; the library verifies a 234 MiB capsule in less than a second;
- opening a capsule needs about its size in memory, in addition to the workspace it replaces;
- recording the provenance of a processing costs the fingerprints of its inputs and results: nothing measurable for a small signal, +29 ms for a 2048×2048 image filtered in 450 ms;
- consulting the provenance of a result takes less than a millisecond, even with 1000 recorded steps; listing 1000 steps in Local replay takes about half a second.

Each processing gets slightly slower as a panel holds more objects, with or without provenance (about 0.1 s in an empty panel, 0.3 s with 1000 objects). DataLab-Web limits capsules to 256 MiB.

Table 1: Accepted file policies, not guaranteed capacities

Consumer	Workspace / manifest
Desktop and library	4 GiB / 64 MiB; the archive has a separate total-size limit
Web	256 MiB / 16 MiB; archive also limited to 256 MiB
Reference registry	No workspace; manifest 8 MiB, review 1 MiB

Each capsule accepts at most 8 attachments of 16 MiB each. Memory can hold several representations of the archive and HDF5 while exporting; a valid file may exceed another edition's policy. The ledger also has JSON volume and structural bounds, even without a fixed activity-count limit. View truncation does not remove records. See the technical limits and measured capacity.

Local replay verifies one activity on copies; it does not reconstruct an entire chain, restore all missing states or provide a workflow for new datasets. Environment versions are described, not reinstalled: a capsule is not an OS with all its dependencies, and a successful comparison is not universal reproducibility between machines.

Fingerprints detect changes but do not authenticate authors; capsules and reviews are unsigned and their fingerprints can be recomputed after editing. Import never runs received macros, notebooks, attachments or Python names; explicit replay uses locally registered, qualified operations. The prescribed profile accepts native supported workspaces, not arbitrary ZIP/HDF5 files: ZIP entries are uncompressed (independent of internal HDF5 compression), and links and encrypted archives are refused. Format checks do not replace a sandbox: HDF5 parsing and dependencies remain trusted code on the Desktop.

Profile 0.1 is experimental and unknown versions are refused. RO-Crate/PROV metadata do not guarantee that any third-party reader can restore or replay DataLab. See the common rules and security analysis. Desktop and the CLI can work offline once installed; the Web downloads its runtime at startup and some optional packages on first use. Computations remain local; neither a registry nor attachments certify the physical acquisition.

3.2 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.

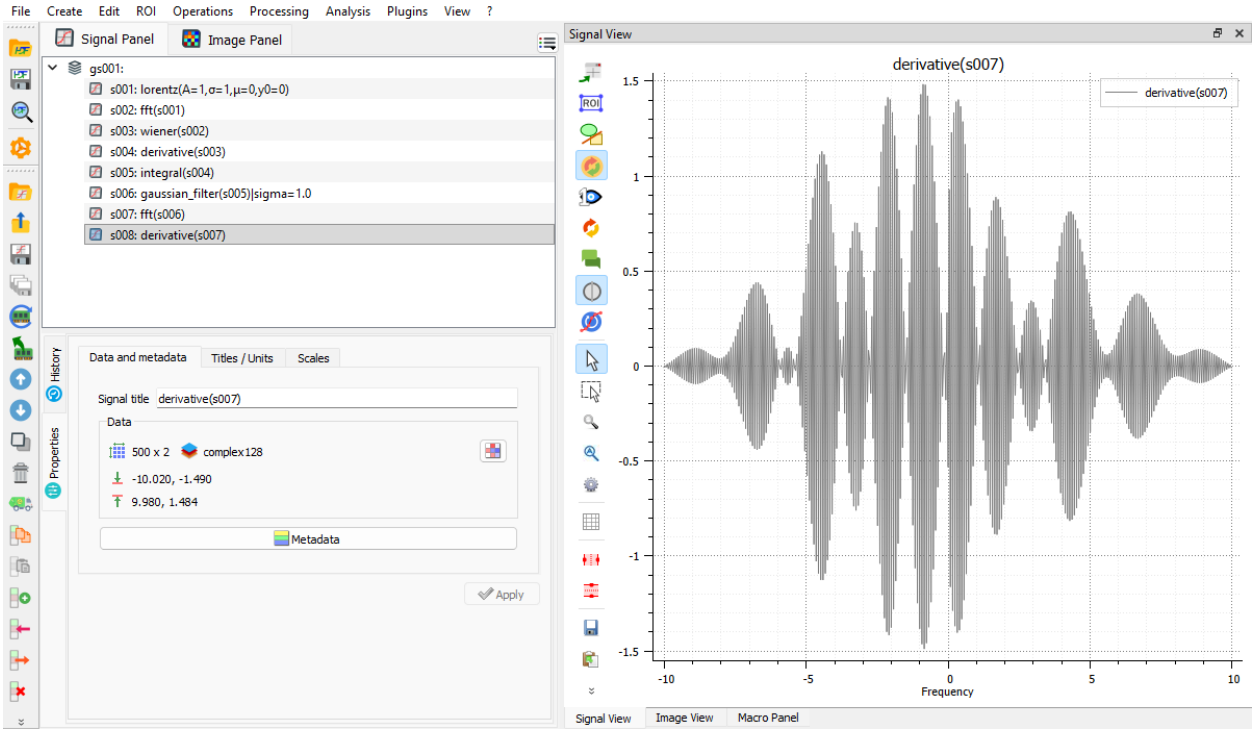


Fig. 13: DataLab main window: Signal processing view

3.2.1 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).
- Export, inspect, replay and compare workspace capsules, i.e. a workspace with the provenance of its processing (see Workspace capsules).
- Edit DataLab preferences (see Settings).
- Start or stop the Web API server, copy its connection information, and check its status (see Web API).
- Quit DataLab.

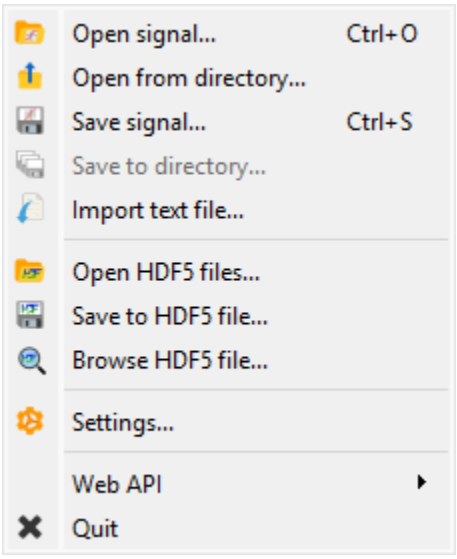


Fig. 14: Screenshot of the "File" menu.

Open signal

Create a new signal from the following supported filetypes:

File type	Extensions
Text files	.txt, .csv
NumPy arrays	.npy
MAT-Files	.mat
HDF5 signal files	.h5sig
FT-Lab files	.sig
MCA files	.mca

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, .txt
NumPy arrays	.npy
MAT-Files	.mat
HDF5 signal files	.h5sig

Save signals to directory

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

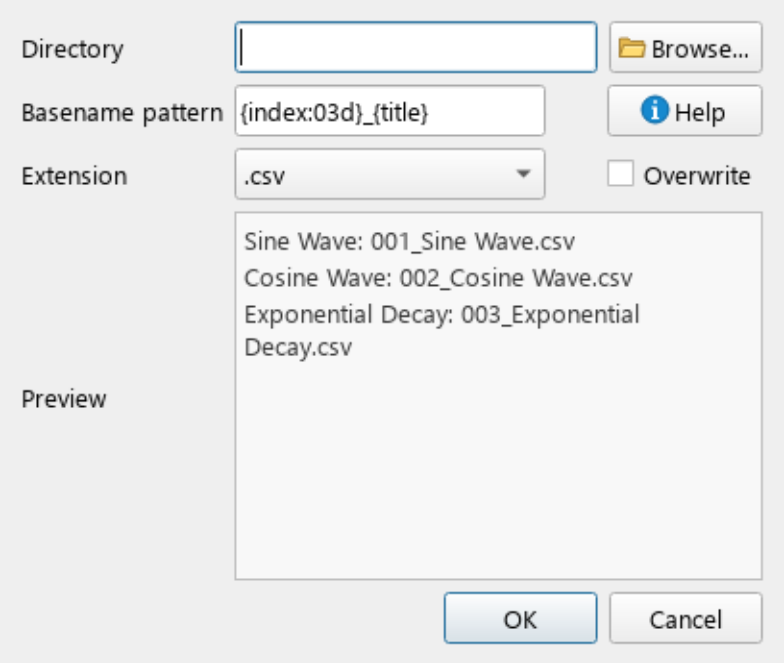


Fig. 15: Save signals to directory dialog.

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
- Basename pattern: Define a pattern for the filenames using Python format strings (the Help button describes the available placeholders)
- 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 (each file name is preceded by the group and the title of the corresponding signal)

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.

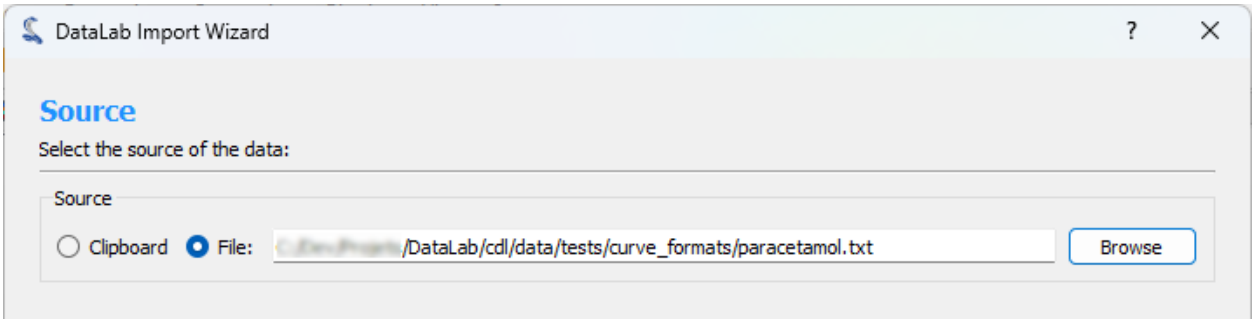


Fig. 16: Step 1: Select the source

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.
- Decimal: The decimal separator (point or comma).
- 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.
- Max. nb of rows: The maximum number of rows to import. If the file contains more rows, they will be ignored.
- Header: The row containing the column names, if any (none, inferred from the data, or first row).
- 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.

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.

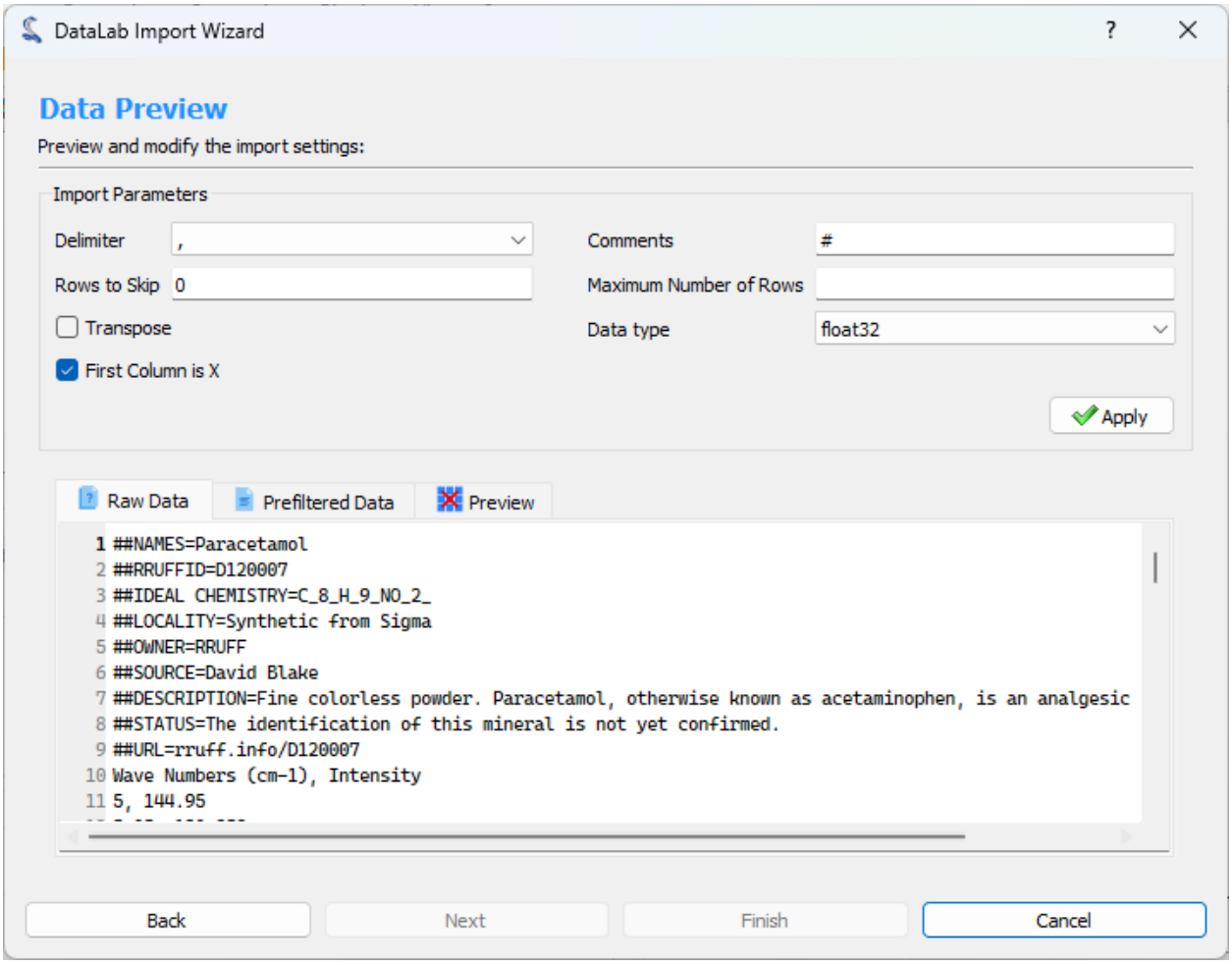


Fig. 17: Step 2: Configure the import

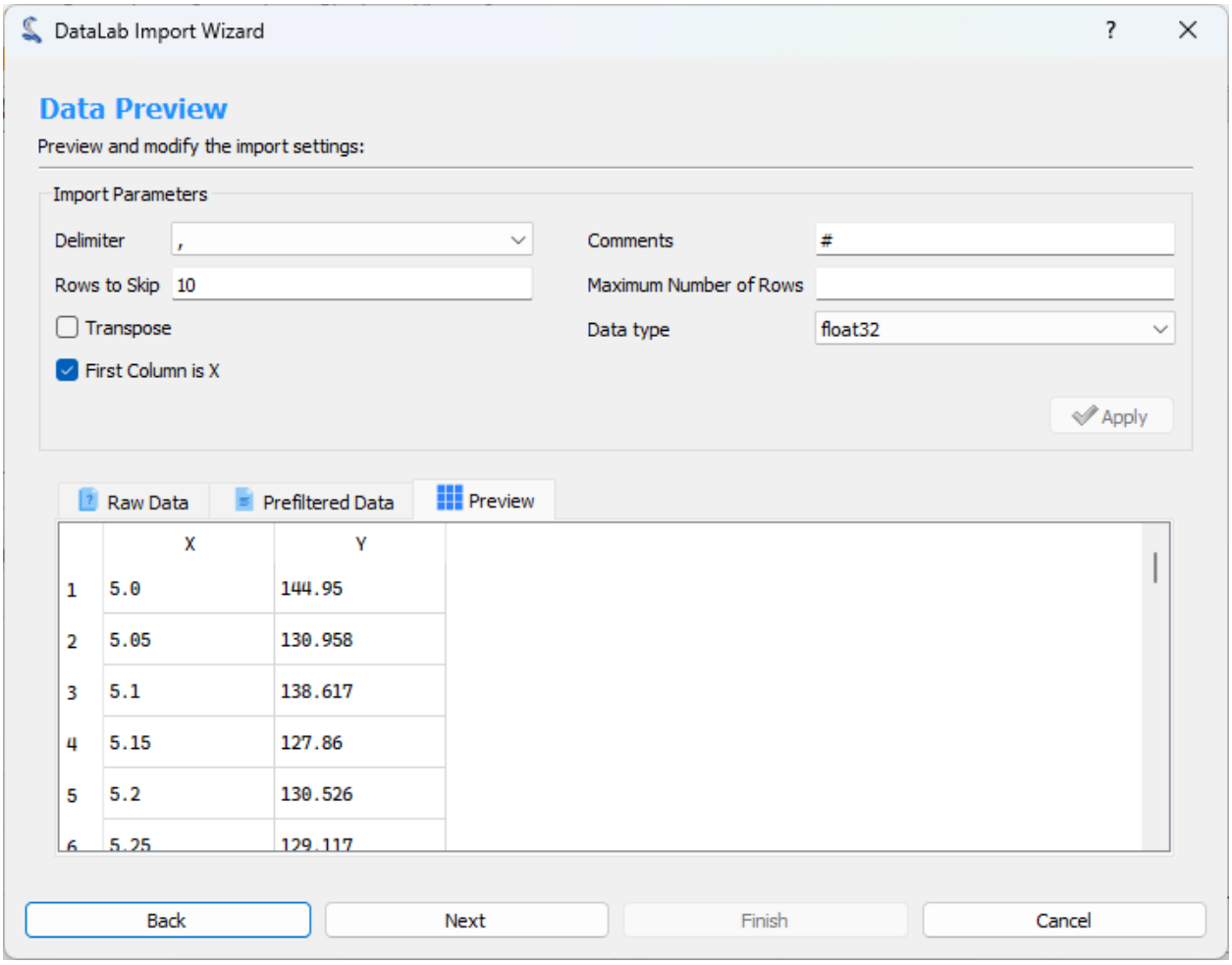


Fig. 18: Step 2: Preview the result

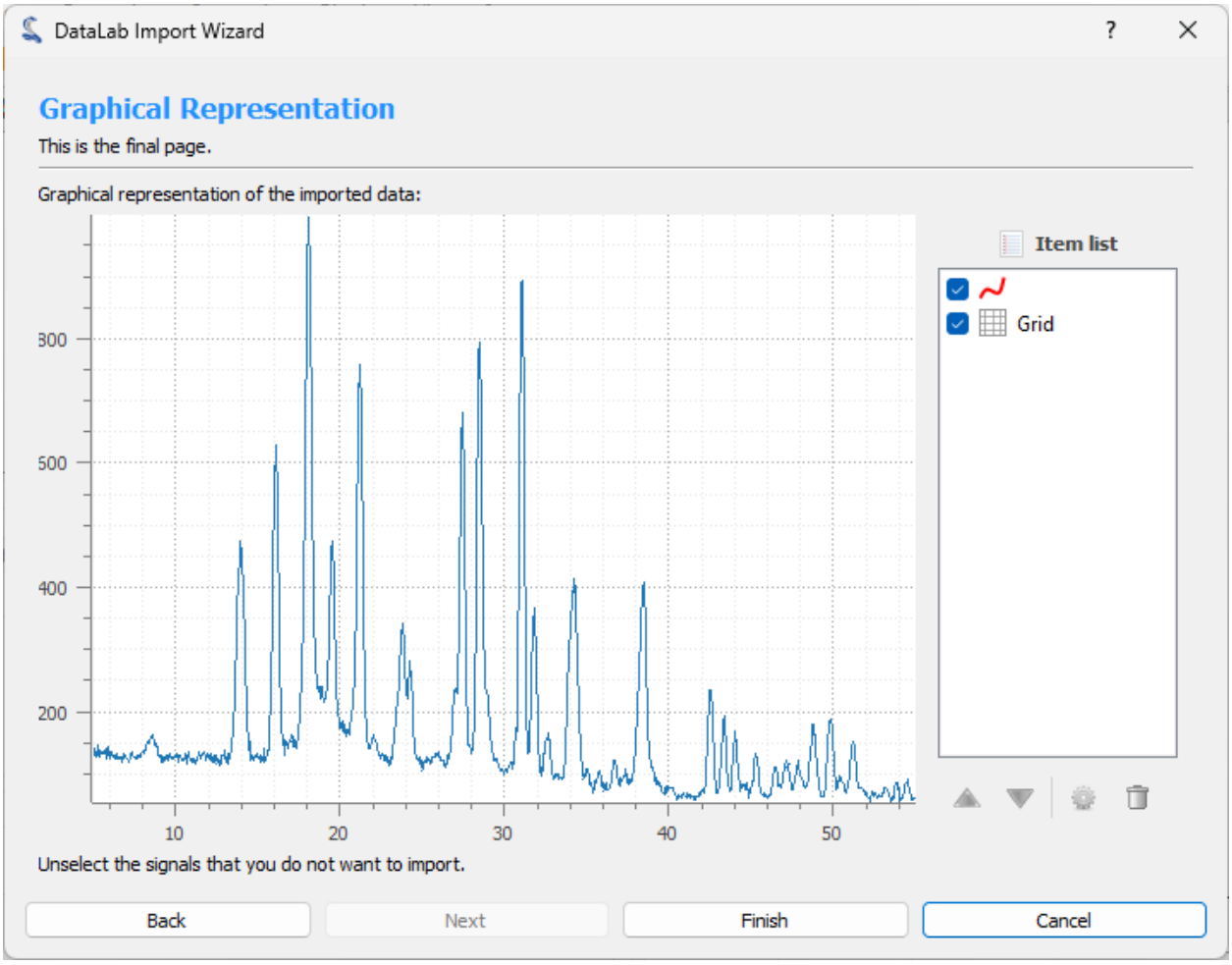


Fig. 19: Step 3: Show graphical representation

3.2.2 Create Signals

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

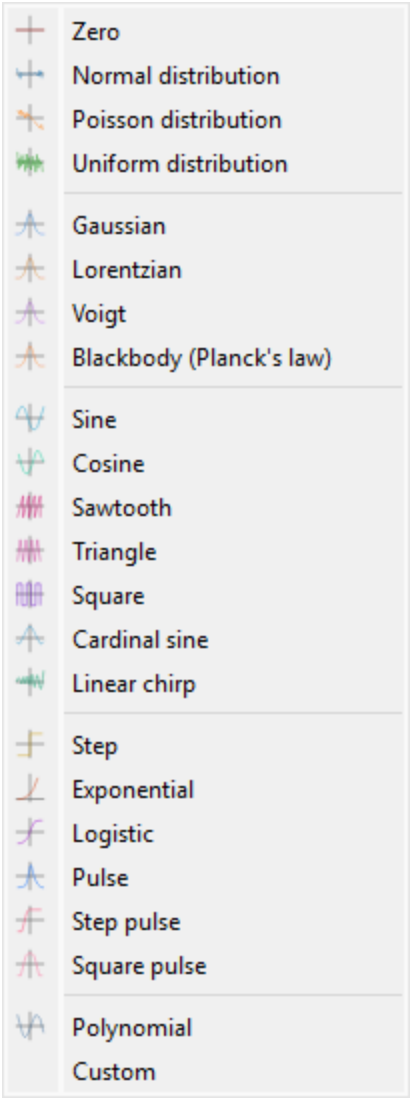


Fig. 20: Screenshot of the “Create” menu.

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).

New signal

Create a new signal from various models:

Icon	Model	Equation
	Zero	$y[i] = 0$

continues on next page

Table 2 – continued from previous page

Icon	Model	Equation
	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	$y = y_0 + A \cdot \exp\left(-\frac{1}{2} \cdot \left(\frac{x-\mu}{\sigma}\right)^2\right)$
	Lorentzian	$y = y_0 + \frac{A}{1 + \left(\frac{x-\mu}{\sigma}\right)^2}$
	Voigt	$y = y_0 + A \cdot \frac{\Re\left(w(z)\right)}{\Re\left(w(j/\sqrt{2})\right)}$ with $w(z) = \exp\left(-z^2\right) \cdot \mathrm{erfc}(-j \cdot z)$ and $z = \frac{x - \mu + j \cdot \sigma}{\sqrt{2} \cdot \sigma}$
	Blackbody (Planck's law)	$y = \frac{2hc^2}{\lambda^5 \left(\exp\left(\frac{hc}{\lambda k T}\right) - 1\right)}$
	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	$y = y_0 + A\left(2(u - \lfloor u \rfloor) - 1\right)$, where $u = fx + \phi/(2\pi)$
	Triangle	$y = y_0 + A\,\mathrm{sawtooth}\left(2\pi f x + \phi, \mathrm{width} = 0.5\right)$
	Square	$y = y_0 + A \begin{cases} 1 & 0 \le u - \lfloor u \rfloor < 1/2 \\ -1 & 1/2 \le u - \lfloor u \rfloor < 1 \end{cases}$, where $u = fx + \phi/(2\pi)$
	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	$y = \begin{cases} A_1 & \text{if } x \le x_0 \\ A_2 & \text{otherwise} \end{cases}$
	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	$y = y_0 + A \begin{cases} 1 & \text{if } x_0 \le x \le x_1 \\ 0 & \text{otherwise} \end{cases}$

continues on next page

Table 2 – continued from previous page

Icon	Model	Equation
		$y = \begin{cases} y_0 & \text{if } x < t_0 \\ y_0 + A \cdot \dfrac{x - t_0}{t_r} & \text{if } t_0 \leq x < t_0 + t_r \\ y_0 + A & \text{if } x \geq t_0 + t_r \end{cases} + \mathcal{N}(0, \sigma_n)$
	Step Pulse	where: • t_0 is the pulse start time, • t_r is the rise time, • σ_n is the noise amplitude
		$y(x) = \begin{cases} y_0 & \text{if } x < t_0 \\ y_0 + A \cdot \dfrac{x - t_0}{t_r} & \text{if } t_0 \leq x < t_0 + t_r \\ y_0 + A & \text{if } t_0 + t_r \leq x < t_1 \\ y_0 + A - A \cdot \dfrac{x - t_1}{t_f} & \text{if } t_1 \leq x < t_1 + t_f \\ y_0 & \text{if } x \geq t_1 + t_f \end{cases} + \mathcal{N}(0, \sigma_n)$
	Square Pulse	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 $d = t_{\text{FWHM}} - \dfrac{t_r + t_f}{2}$ from the full width at half maximum t_{FWHM} **Warning** The duration of the plateau d should not be negative.
	Polynomial	$y = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_5 \cdot x^5$
	Custom	Manual input of X and Y values

For Gaussian, Lorentzian and Voigt models, A is the signed peak height above the baseline y_0, expressed in the signal's Y unit.

For the sine, cosine, sawtooth, triangle, square and cardinal sine signals, the phase ϕ is entered in degrees in the creation dialog, while the formulas above use it in radians. The initial phase ϕ_0 of the linear chirp is entered in radians.

The cardinal sine uses the unnormalized definition $\text{sinc}(u) = \dfrac{\sin(u)}{u}$, with $\text{sinc}(0) = 1$, as the 2D sinc image: when $\phi = 0$, its peak is at $x = 0$ and its first zeros are at $x = \pm\dfrac{1}{2f}$.

The blackbody signal is the spectral radiance per unit wavelength of a blackbody at temperature T: wavelengths λ are expressed in meters and the spectral radiance in W·sr^{-1}·m^{-3}. These axis titles and units are set by default, and may be changed in the creation dialog.

3.2.3 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).

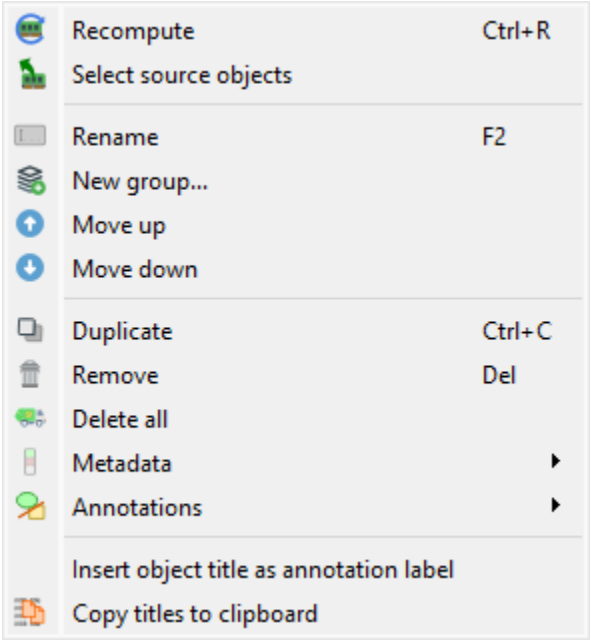


Fig. 21: Screenshot of the "Edit" menu.

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.

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.

It also refreshes analysis results (statistics, FWHM, centroid, peak/contour detection, etc.) using their stored parameters. Analysis results are not recomputed automatically when you modify a region of interest, the data, or object properties: existing results are left untouched until you explicitly trigger this action, giving you full control over when analyses are refreshed.

> **Note**
>
> This action is enabled as soon as a signal or a group is selected, but it only applies to signals that have stored processing

parameters (created through processing operations) or stored analysis parameters (analysis operations previously run on them). When none of the selected signals has such parameters, an information message is displayed.

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 enabled when exactly one signal is selected. When that signal has no source object references, an information message is displayed.

Metadata

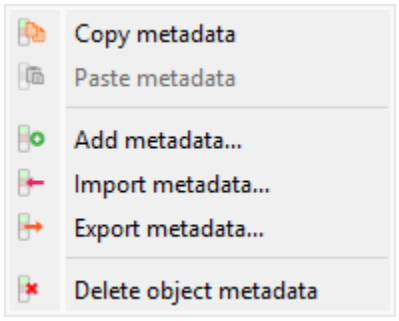


Fig. 22: Screenshot of the "Metadata" submenu.

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.

When pasting metadata, a dialog box allows you to select which metadata categories to keep from the clipboard. Annotations are copied and pasted separately:

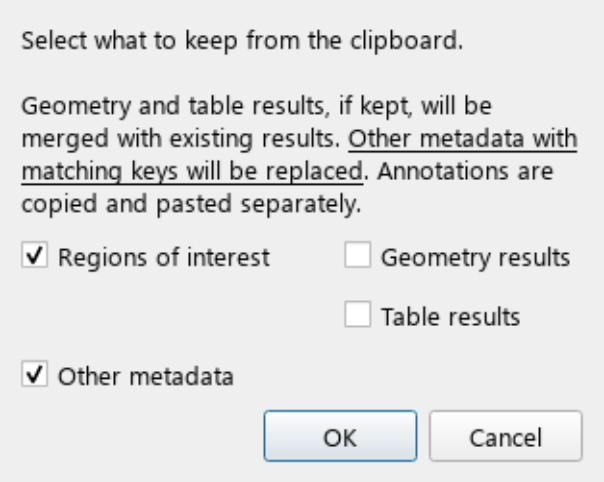


Fig. 23: Screenshot of the "Paste metadata" dialog.

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 object metadata

When deleting metadata using the "Delete object 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.

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 (letters, digits, _, . and -, e.g. a plugin key such as `plugin.org.example.my-plugin.gain`)
- Known keys: Pick a key already present on the selected signals, or a key expected by the methods of the installed application plugins, to copy it into the metadata key

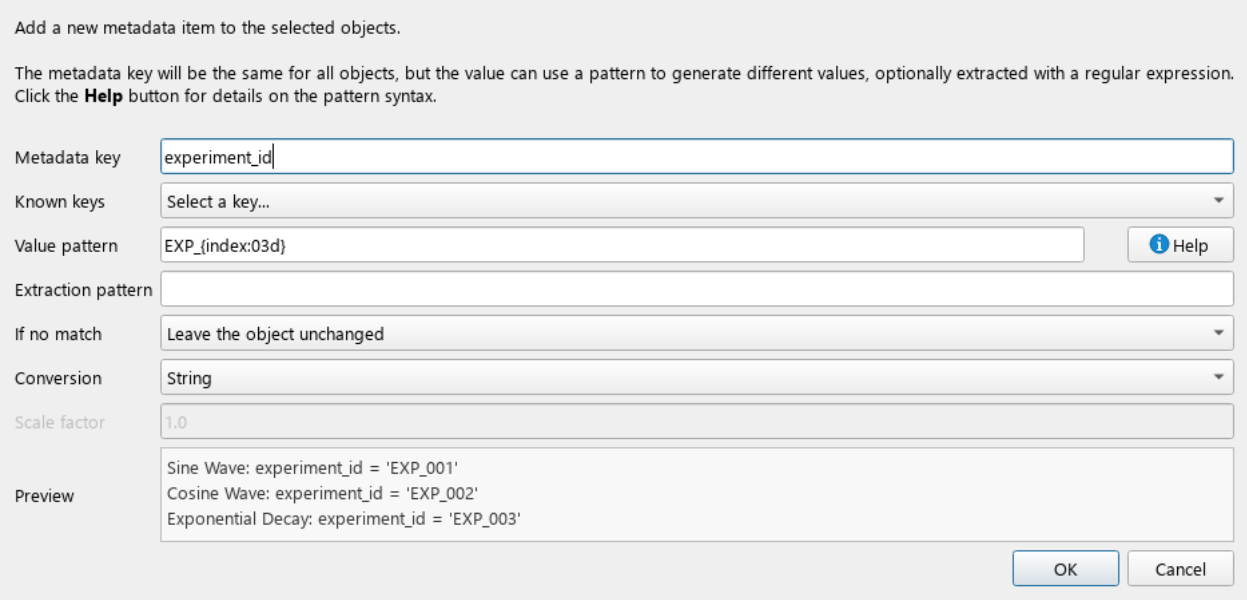


Fig. 24: Add metadata dialog.

- Value pattern: Define a pattern for the metadata value using Python format strings
- Extraction pattern: Optionally, extract the value from the formatted text with a Python regular expression: its first group, or the whole match if it has no group, becomes the value
- If no match: Leave signals without a match unchanged (default), or report an error
- Conversion: Choose how to store the value (string, float, integer, or boolean)
- Scale factor: Multiply numeric values, e.g. `0.001` to convert milliseconds to seconds
- Preview: See how the metadata will be added to each selected signal, and which signals are left unchanged

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
- Number shots from their titles: key=``shot``, pattern=``{title}``, extraction=``shots*(d+)``, conversion=integer → Sets `shot=42` on a signal titled "CH1 shot 042"

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.

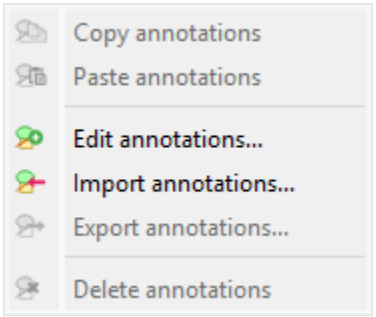


Fig. 25: Screenshot of the "Annotations" submenu.

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:

- "Insert object title as annotation label": 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 (titles of processed signals refer to their sources by the first 8 characters of their unique identifier, prefixed by "g" for groups):

```
Raw data
    lorentz(A=1,σ=1,μ=0,y0=0)
    derivative(511f1709)
    wiener(241270f6)
derivative(g25fb9d07)
    derivative(511f1709)
    derivative(241270f6)
    derivative(fedb39d5)
```

3.2.4 Regions Of Interest (ROI)

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

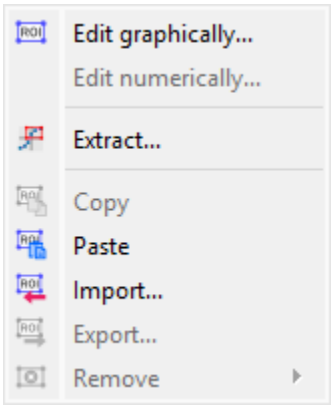


Fig. 26: Screenshot of the "ROI" menu.

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.
- The “Processing” menu actions are performed only on the ROI if one is defined (except if the operation changes the number of points, like interpolation or resampling, or if the destination signal data type is different from the source’s, like in the Fourier analysis features).
- The “Analysis” menu actions generally analyze each ROI if defined. Statistics also include a whole-signal row, identified by ROI index `-1`.

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

> **Note**
>
> On the plot, each ROI is drawn as a translucent area clipped to the signal curve (with a baseline at `y = 0`, or at the bottom of the canvas on a logarithmic Y axis), instead of a rectangle spanning the whole canvas height. Successive ROIs use different colors, taken from a cyclic ten-color palette: the color is derived from the `ROI<n>` title index, so a given ROI keeps the same color across deletions, reorderings and editor reopenings. Only the coordinates are stored: these visual properties are recomputed each time the ROIs are drawn.

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.

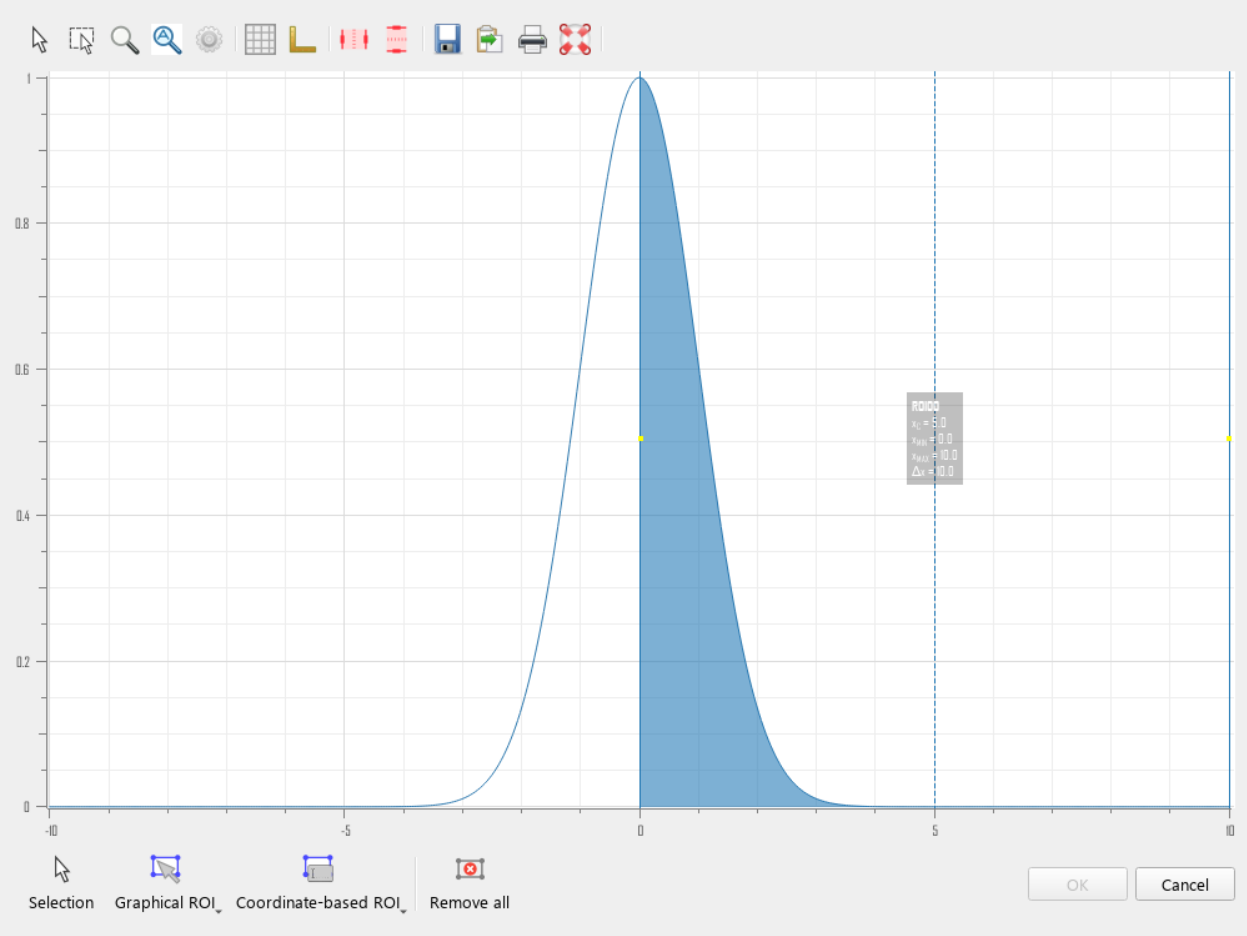


Fig. 27: A signal with an ROI.

- "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" : remove a single ROI (one entry per ROI of the selected signal) or all defined ROIs for the selected signals ("Remove all").

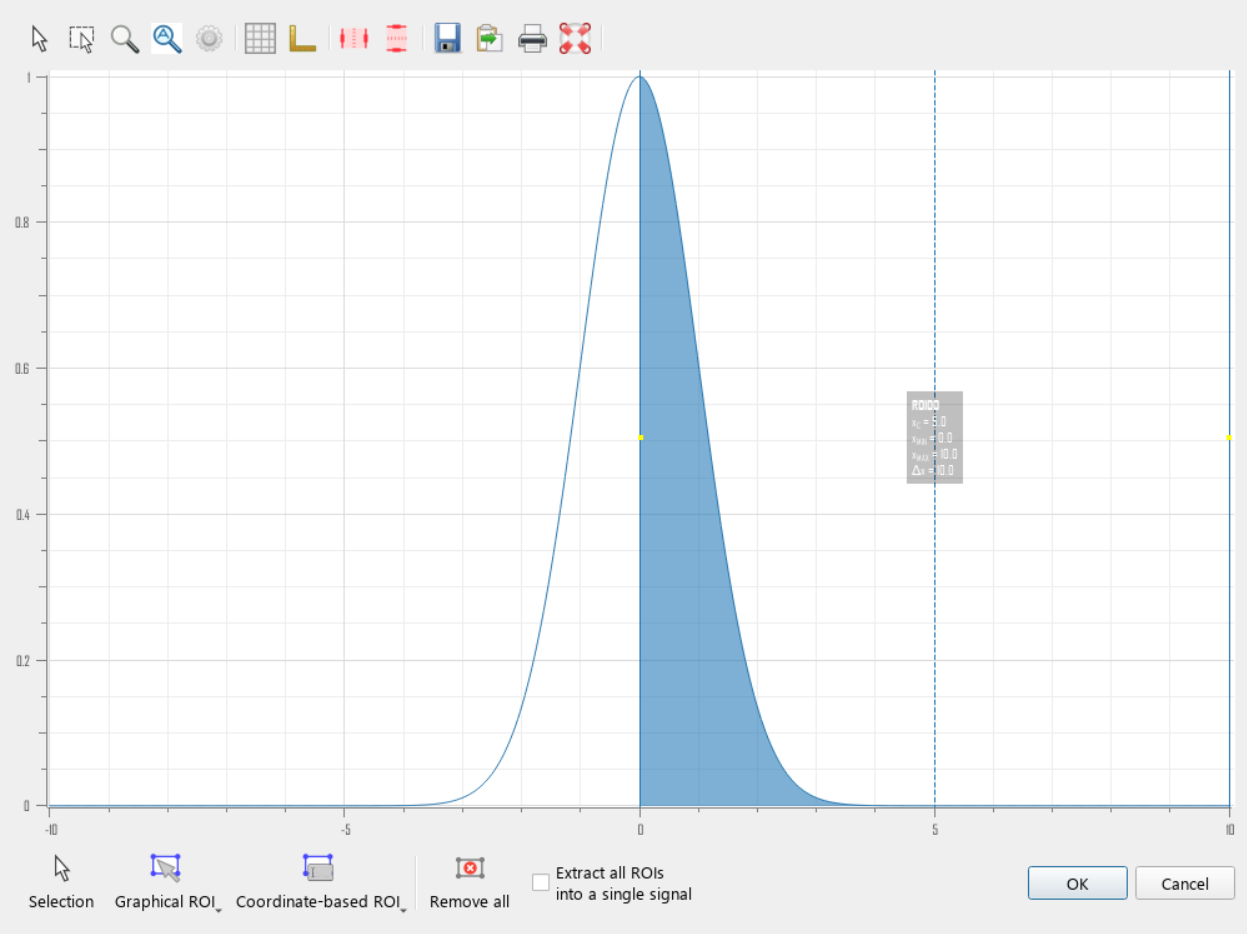


Fig. 28: ROI extraction dialog: the ROI is defined by moving the position and adjusting the width of an horizontal range.

3.2.5 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.

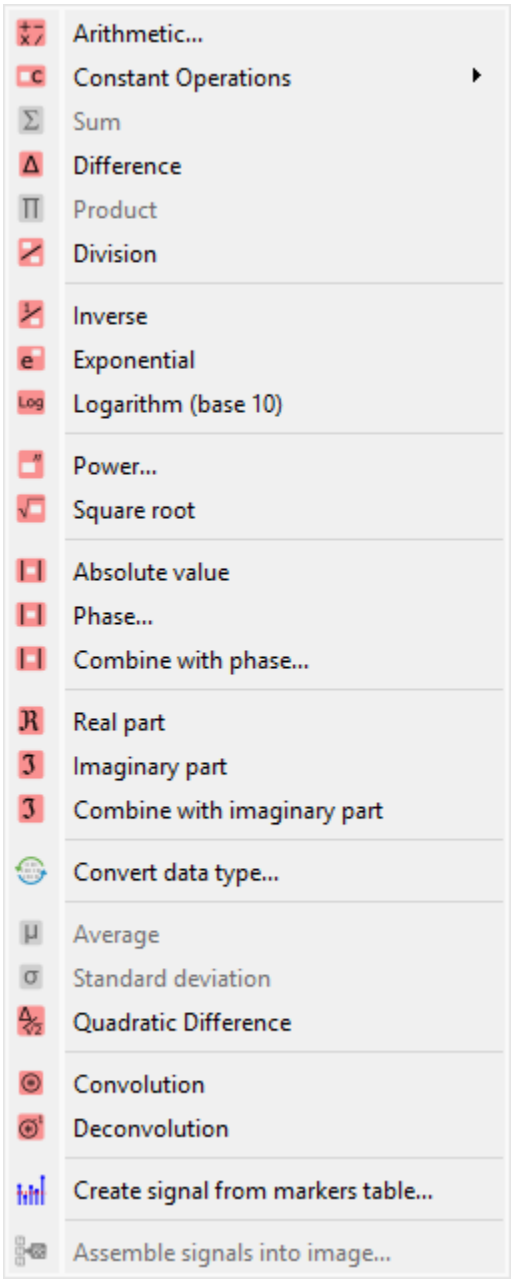


Fig. 29: Screenshot of the “Operations” menu.

The “Operations” menu allows you to perform various operations on the selected signals, such as arithmetic operations, mathematical functions, statistics between signals, or convolution.

Arithmetic

The “Arithmetic” action creates a new signal which is the result of an arithmetic operation between each selected signal and a second signal (selected in a dialog box):

$$y = (y_1 \odot y_2) \times a + b$$

where y_1 is the selected signal, y_2 is the second signal, $\odot$ is the Operator ($+$, $-$, $\times$ or $/$), a is the Factor and b is the Constant. The Operation field shows the resulting expression. By default, the result is converted to the data type of the selected signal.

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	$y_k = \|y_{k-1}\|$
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	$y_M = \sqrt{\frac{1}{M}\sum_{k=0}^{M-1} (y_k - \bar{y})^2}$
Quadratic Difference	$y_2 = \frac{y_1 - y_0}{\sqrt{2}}$

Other mathematical functions

Function	Description
Power	$y_k = y_k^n$
Square root	$y_k = \sqrt{y_k}$
Assemble signals into image	Create a 2D image by assembling selected 1D signals as rows or columns, with optional normalization.
Create signal from markers table…	Build a sticks signal from an XY-markers table result attached to the selected signal — typically produced by Analysis → Extract peak positions. When several XY-markers tables are stored on the signal, a dialog lets you pick the source table.

3.2.6 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.

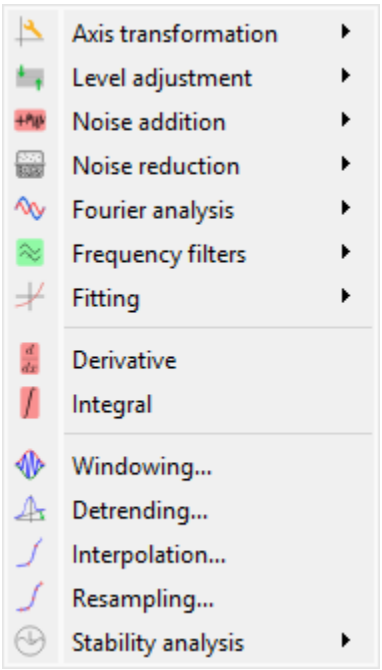


Fig. 30: Screenshot of the “Processing” menu.

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.

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.

Replace X by other signal's Y

Create a new signal by combining two signals, using Y values from the second signal as X coordinates for the first signal's Y values.

This feature is particularly useful for calibration scenarios, such as:

- Applying wavelength calibration to spectroscopy data
- Using a calibrated reference signal to rescale measurement data
- Combining signals where one contains calibration points

The two signals must have the same number of points. The operation directly uses the Y arrays without interpolation. With Sigima 1.4.1 or later, if the first signal has a region of interest, only its samples within this region are kept, and the result has no region of interest.

Input	Description
Signal 1	Provides the Y data for the output signal
Signal 2	Provides the Y data that becomes the X coordinates of the output signal

X-Y mode

Create a new signal by simulating the X-Y mode of an oscilloscope.

This operation plots one signal as a function of another by:

1. Finding the overlapping X range between both signals
2. Resampling both signals onto a common X grid within that overlap
3. Using the resampled Y values as coordinates (first signal's Y as X, second signal's Y as Y)

With Sigima 1.4.1 or later, signals sharing the same X grid are paired without interpolation; otherwise, the common grid is uniform, with the smaller of the two input point counts. Non-overlapping domains and signals with different X units are rejected. If the first signal has a region of interest, only the points within it are kept, and the result has no region of interest. The X and Y labels and units of the result are those of the Y axes of the first and second signals. Earlier Sigima versions may extend an operand beyond its original domain in DataLab.

> **Note**
>
> This is different from "Replace X by other signal's Y" as it performs interpolation to handle signals with different X arrays.

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	$y_1 = \frac{y_0}{\sum_{n=0}^{N-1} y_0[n]}$
Energy	$y_1 = \frac{y_0}{\sqrt{\sum_{n=0}^{N-1} \lvert y_0[n] \rvert^2}}$
RMS	$y_1 = \frac{y_0}{\sqrt{\frac{1}{N}\sum_{n=0}^{N-1} \lvert y_0[n] \rvert^2}}$

Here N is the number of samples, assuming no NaNs. The implemented sum and mean ignore NaNs.

Clipping

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

Replace special values

Create a new signal where `NaN`, `+Inf` and `-Inf` samples are replaced using a configurable strategy per target value.

Each of the three targets (`NaN`, `+Inf`, `-Inf`) is handled independently. For each target the user selects one of the following strategies (or Do nothing to leave the target untouched):

Strategy	Description
Do nothing	Leave the target values untouched.
Replace with zero	Replace by `0`.
Replace with constant	Replace by a user-defined constant value.
Replace with minimum / maximum	Replace by the minimum / maximum of the finite samples.
Replace with mean / median	Replace by the mean / median of the finite samples.
Delete points	Remove the affected samples from the signal (the X array is also shrunk). A warning is emitted if the signal was uniformly sampled.
Forward fill / Backward fill	Propagate the last (resp. next) valid value.
Interpolation: Linear / Spline / Quadratic / Cubic / PCHIP	Reconstruct missing samples by interpolating the surrounding valid samples.
N-neighbor minimum / maximum / mean / median	Replace by the corresponding statistic computed on the `2N+1` neighboring samples around each affected position.

Before opening the standard parameter form, the dialog displays a count of each target found in the signal (e.g. NaN: 17 (1.7 %)). When a neighbor strategy is selected, a small preview shows the kernel that will be used.

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)
Location	Where to add the zeros: 'append' (after the signal), 'prepend' (before the signal) or 'both' (half before, half after)
Number of points	Number of points to add (user-defined if strategy is "custom", computed automatically otherwise)

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, greater than or equal to the signal length (e.g. 1000 → 1024)
double	Double the length (e.g. 1024 → 2048)
triple	Triple the length (e.g. 1024 → 3072)
custom	Custom number of points to add (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)	$y_1 = \lvert\text{FFT}\,(y_0)\rvert$ or $y_1 = 20\log_{10}(\lvert\text{FFT}\,(y_0)\rvert)$ (dB)
Phase spectrum	Phase of the FFT expressed in degrees, using numpy.angle	$y_1 = \angle\,\text{FFT}\,(y_0)$
Power spectral density (PSD)	Optional: output in decibels (dB). PSD is estimated using Welch's method (see scipy.signal.welch)	$y_1 = \text{PSD}\,(y_0)$ or $y_1 = 10\log_{10}(\text{PSD}\,(y_0))$ (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 (Interactive fitting submenu), which shows the formula of the model below the fit parameters. Most models are available in both modes, except Multi-Gaussian and Multi-Lorentzian fits (interactive mode only) and Sigmoid fit (automatic mode only).

The fitted signal stores the model parameters in its metadata (`metadata["fit_params"]`), under the names given in the last column of the table below:

Model	Equation	Stored parameters
Linear	$y = a \cdot x + b$	`a`, `b`
Polynomial	$y = a \cdot x^n + b \cdot x^{n-1} + c \cdot x^{n-2} + \ldots$	`a`, `b`, `c`, ...
Gaussian	$y = y_0 + A \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma}\right)^2\right)$	A: `amplitude`, σ: `sigma`, x_0: `x0`, y_0: `y0`
Lorentzian	$y = y_0 + \frac{A}{1 + \left(\frac{x - x_0}{\sigma}\right)^2}$	A: `amplitude`, σ: `sigma`, x_0: `x0`, y_0: `y0`
Voigt	$y = y_0 + A \cdot \frac{\Re\left(w(z)\right)}{\Re\left(w(j/\sqrt{2})\right)}$ with $w(z) = \exp(-z^2)\,\mathrm{erfc}(-jz)$ and $z = \frac{x - x_0 + j\sigma}{\sqrt{2}\sigma}$	A: `amplitude`, σ: `sigma`, x_0: `x0`, y_0: `y0`
Multi-Gaussian	$y = y_0 + \sum_{i=1}^{N} A_i \exp\left(-\frac{1}{2}\left(\frac{x - x_{0,i}}{\sigma_i}\right)^2\right)$	A_i: `amplitude_i`, σ_i: `sigma_i`, $x_{0,i}$: `x0_i`, y_0: `y0`
Multi-Lorentzian	$y = y_0 + \sum_{i=1}^{N} \frac{A_i}{1 + \left(\frac{x - x_{0,i}}{\sigma_i}\right)^2}$	A_i: `amplitude_i`, σ_i: `sigma_i`, $x_{0,i}$: `x0_i`, y_0: `y0`
Planck	$y = y_0 + A \cdot \left(\frac{x}{x_0}\right)^{-5} \cdot \left(\exp\left(\frac{5x_0}{\sigma \cdot x}\right) - 1\right)^{-1}$ where x_0 and σ are scale and temperature factors (see the note below)	A: `amp`, x_0: `x0`, σ: `sigma`, y_0: `y0`
Two half Gaussians	$y = y_{0,L} + A_L \cdot \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma_L}\right)^2\right)$ if $x < x_0$ $y = y_{0,R} + A_R \cdot \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma_R}\right)^2\right)$ otherwise	A_L: `amp_left`, A_R: `amp_right`, σ_L: `sigma_left`, σ_R: `sigma_right`, x_0: `x0`, $y_{0,L}$: `y0_left`, $y_{0,R}$: `y0_right`
Piecewise exponential (raise-decay)	$y = y_0 + a_L \cdot \exp\left(b_L \cdot x\right)$ if $x < x_c$ $y = y_0 + a_R \cdot \exp\left(b_R \cdot x\right)$ otherwise	x_c: `x_center`, a_L: `a_left`, b_L: `b_left`, a_R: `a_right`, b_R: `b_right`, y_0: `y0`
Exponential	$y = y_0 + a \exp\left(b \cdot x\right)$	`a`, `b`, y_0: `y0`
Sinusoidal	$y = y_0 + A \sin\left(2\pi f \cdot x + \phi\right)$	A: `amplitude`, f: `frequency`, ϕ: `phase` (in radians), y_0: `offset`
Cumulative Distribution Function (CDF)	$y = y_0 + A \,\mathrm{erf}\left(\frac{x - \mu}{\sqrt{2}\sigma}\right)$	A: `amplitude`, μ: `mu`, σ: `sigma`, y_0: `baseline`
Sigmoid	$y = y_0 + \frac{A}{1 + \exp\left(-k\left(x - x_0\right)\right)}$	A: `amplitude`, `k`, x_0: `x0`, y_0: `offset`

For Gaussian, Lorentzian and Voigt models, A is the signed peak height above the baseline y_0, expressed in the signal's Y unit.

> **Note**
>
> The Planck model is over-parameterized: it is invariant under $x_0 \to k x_0$, $\sigma \to k\sigma$, $A \to A/k^5$. Only the ratio x_0/σ and the product $A \cdot x_0^5$ are determined by the data, so the individual fitted values must not be interpreted on their own. In particular the emission peak is located at approximately $1.007\, x_0/\sigma$, not at x_0.

The physical quantities are derived from these combinations. With wavelengths in meters, the model matches Planck's law (spectral radiance per unit wavelength) for:

- a temperature $T = \dfrac{hc\,\sigma}{5k_B\,x_0} \approx 2.878 \times 10^{-3}\,\dfrac{\sigma}{x_0}$ K,
- an emissivity (or any other scale factor of the measurement) $\varepsilon = \dfrac{A\,x_0^5}{2hc^2}$, when the signal is a spectral radiance in W·sr^{-1}·m^{-3}.

For example, fitting the blackbody signal created at 293 K gives back $T = 293$ K and $\varepsilon = 1$.

Multi-peak fit parameters

For Multi-Gaussian and Multi-Lorentzian fits, the peak-detection dialog first selects the center $x_{0,i}$ of each component. These positions remain fixed: the interactive fit dialog only exposes controls for each `amplitude_i` and `sigma_i` parameter and for the common `y0` baseline.

When the fit is accepted, the generated signal stores the complete model under `metadata["fit_params"]`, including every fixed `x0_i` center. The metadata may be inspected with the Metadata button or exported to a JSON `.dlabmeta` file using the metadata import/export actions.

The exported parameters are sufficient to evaluate the theoretical model on any X axis outside DataLab:

```
import numpy as np
from sigima.tools.signal import fitting

fit_params = fitted_signal.metadata["fit_params"]
x = np.linspace(xmin, xmax, size)
y_theoretical = fitting.evaluate_fit(x, **fit_params)
```

Evaluate fit

Apply a previously computed fit to a signal. This feature uses the fit parameters stored in the signal metadata (from a prior fitting operation) to generate the fitted curve. This is useful to evaluate the fit on a different X range or on a different signal.

Derivative

Create a new signal which is the derivative of each selected signal. Based on numpy.gradient.

Integral

Create a new signal which is the integral of each selected signal. Based on scipy.integrate.cumulative_trapezoid.

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	Piecewise quadratic interpolation using SciPy's interp1d function with Sigima 1.4.1 or later; earlier versions fit a single quadratic polynomial
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 variance	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.

3.2.7 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.

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 $1/e^2$.

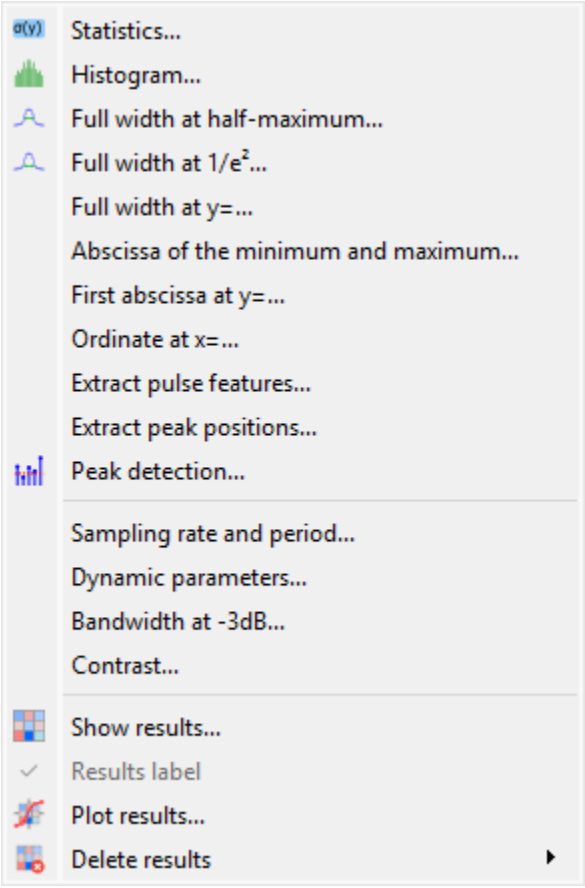


Fig. 31: Screenshot of the “Analysis” menu.

> **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. The "Histogram" feature is an exception: it creates a new signal from each selected signal.

Statistics

Compute statistics on selected signal and show a summary table.

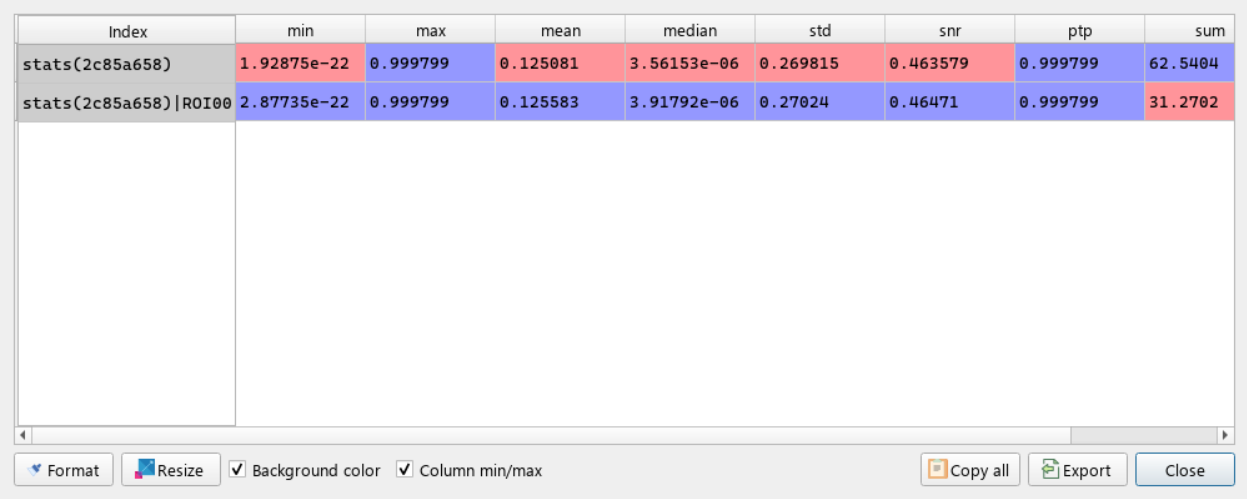


Fig. 32: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

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

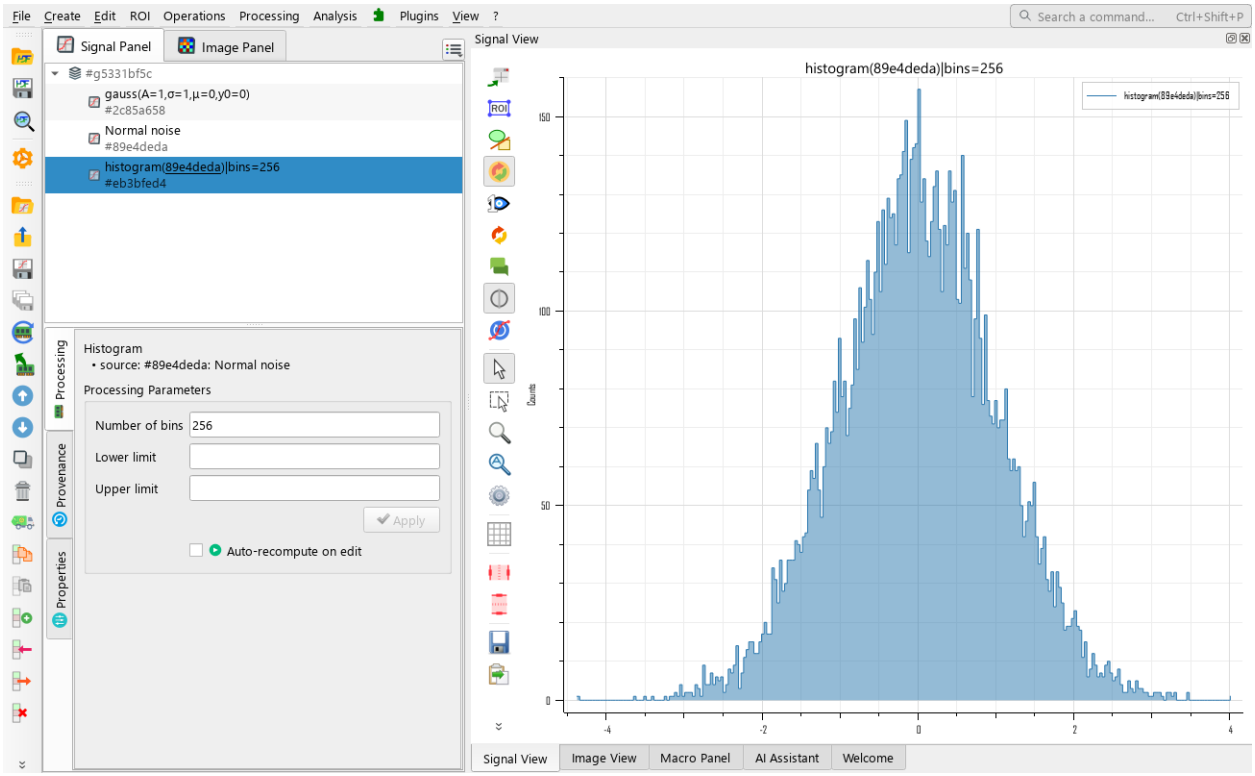


Fig. 33: Example of histogram.

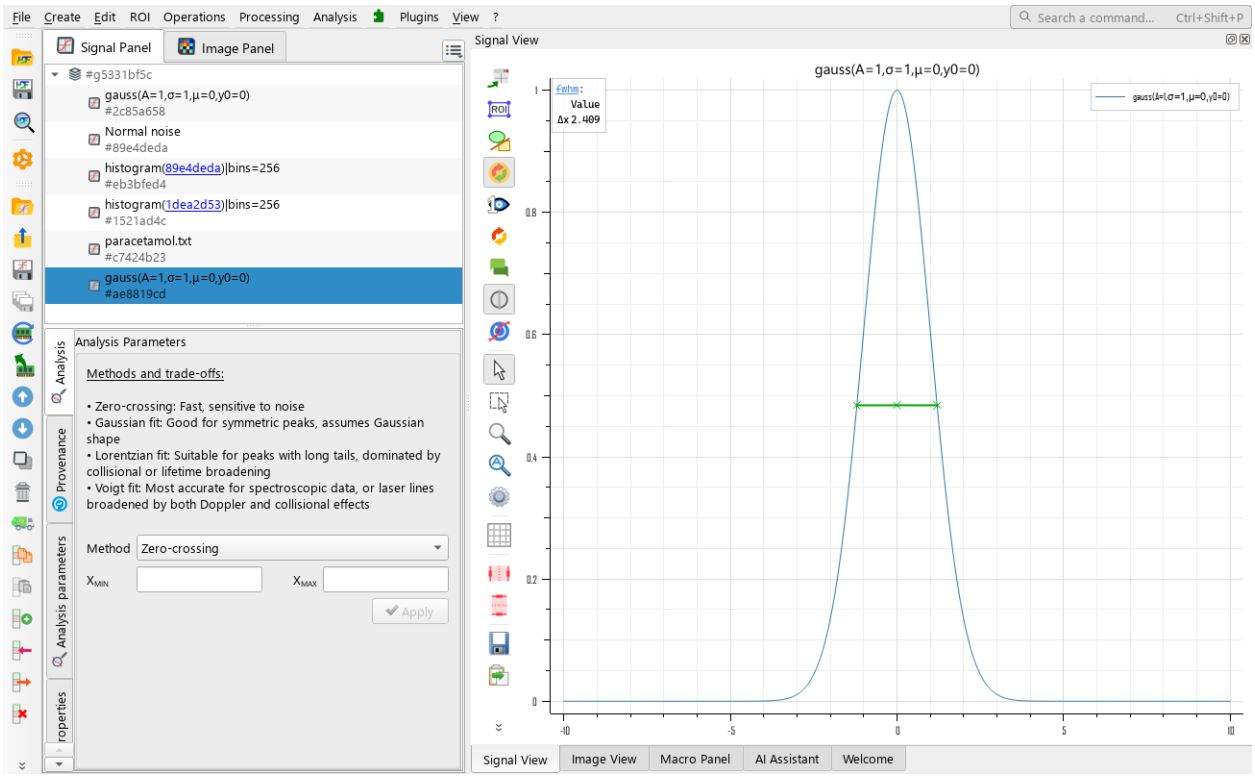


Fig. 34: The computed result is displayed as an annotated segment.

> **Note**
>
> The Zero-crossing method (default) estimates the baseline and the amplitude from the data: it is robust to noise, but it overestimates the width of smooth peaks, e.g. by about 2.5 % to 7 % for a noise-free Gaussian peak, depending on the baseline level. For a peak of known shape, prefer the corresponding fitting method (Gauss, Lorentz or Voigt).

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.

Full width at y=...

Compute the full width of the selected signal at a given ordinate value, chosen by moving a horizontal cursor in a dialog box. The computed result is displayed as an annotated segment.

Abscissa of the minimum and maximum

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

First abscissa at y=...

Compute the abscissa at a given ordinate value for the selected signal (using linear interpolation), chosen by moving a horizontal cursor in a dialog box. If there is no solution, the displayed result is NaN. If there are multiple solutions, the displayed result is the first one.

Ordinate at x=...

Compute the ordinate at a given abscissa value for the selected signal (using linear interpolation), chosen by moving a vertical cursor in a dialog box. If the abscissa is outside the signal range, the displayed result is NaN.

Peak detection

DataLab offers two complementary ways to detect peaks. Both produce the same result: an XY-markers `TableResult` attached to the signal (visible from Analysis Show results and overlayed on the curve).

Entry point	Description
Extract peak positions	Parametric entry: fill the standard parameter dialog (threshold, minimum distance) and run.
Peak detection…	Interactive entry: opens a dedicated dialog where the threshold is adjusted by moving a horizontal marker and the minimum distance by a slider. Detected peaks update live as vertical markers.

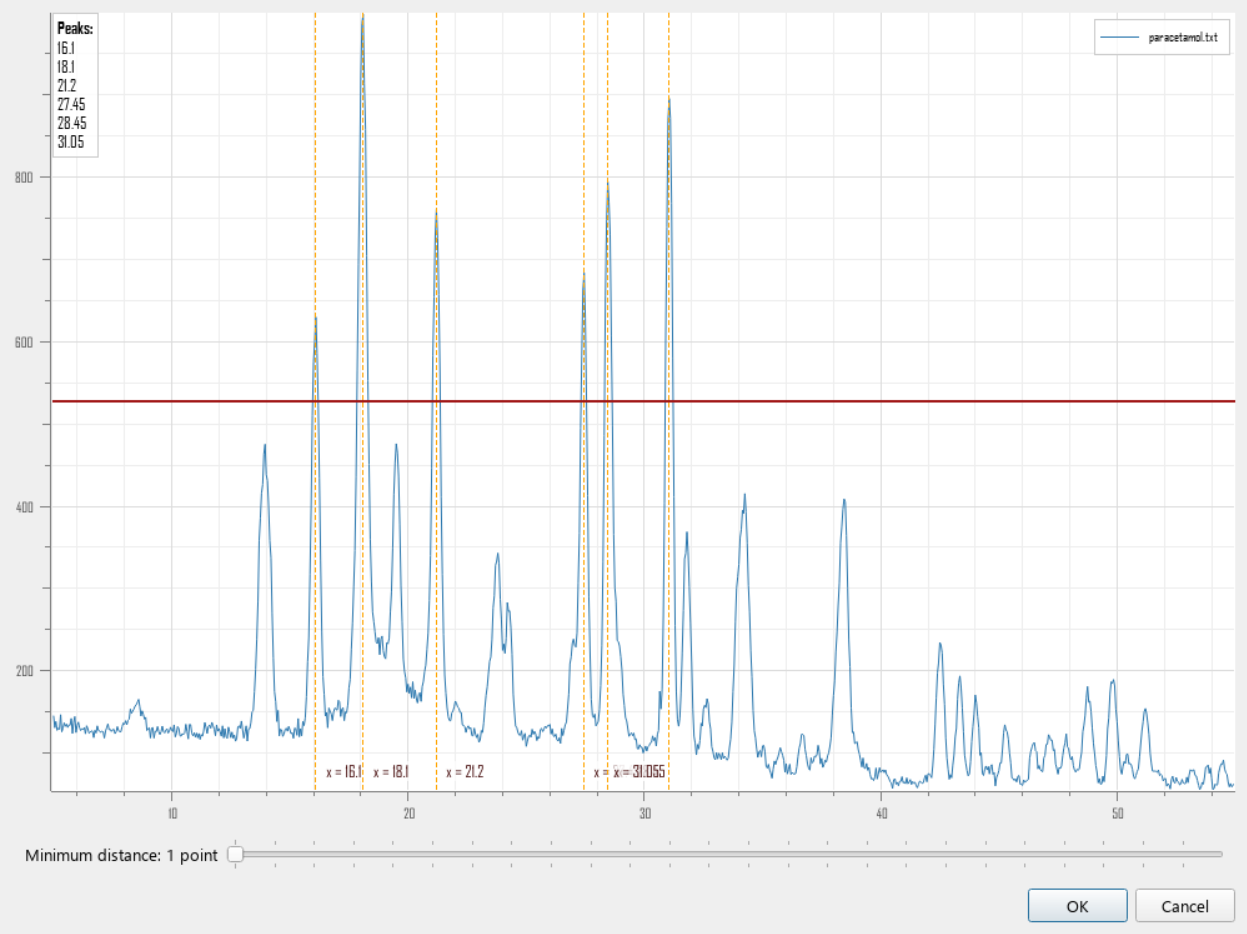


Fig. 35: Peak detection dialog: threshold is adjustable by moving the horizontal marker, peaks are detected automatically (see vertical markers with labels indicating peak position).

> **Tip**
>
> To rebuild the historical sticks signal from the detected peaks, use Operations Create signal from markers table… on the selected signal.

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.

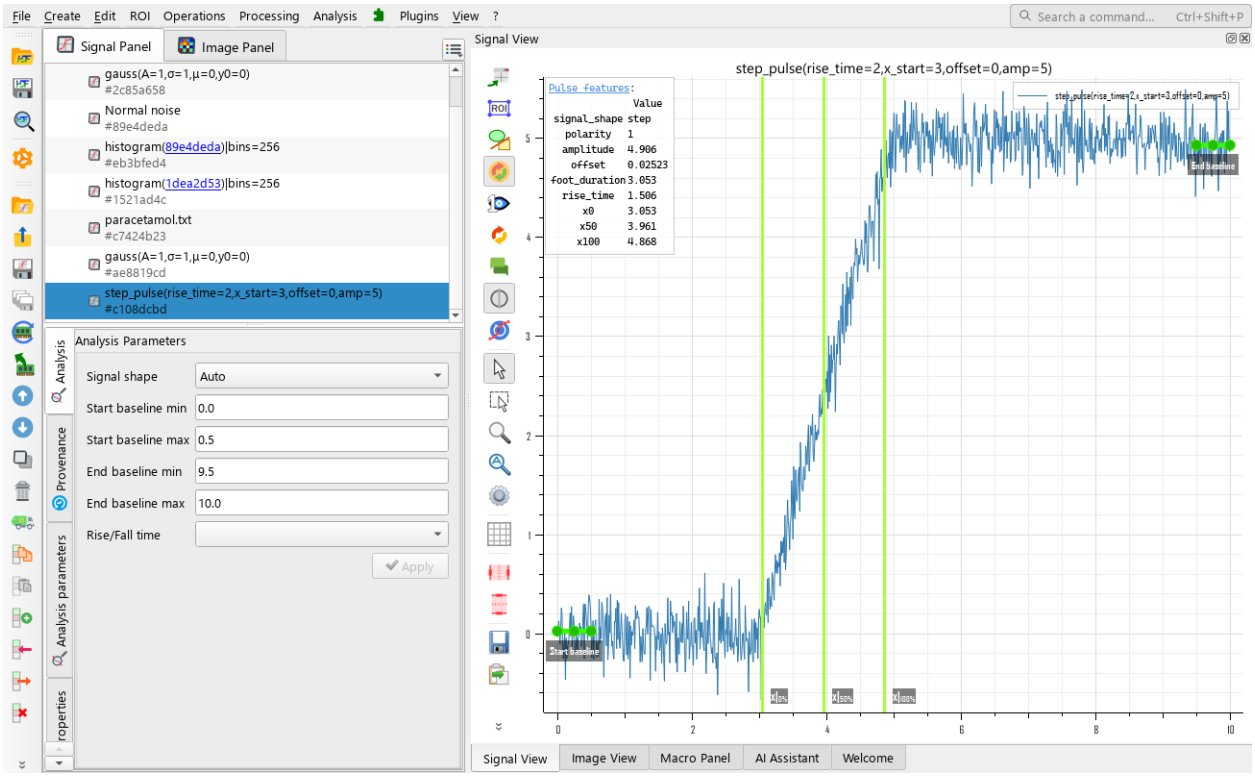


Fig. 36: Pulse features results as displayed in the Signal View.

Key Capabilities:

- Automated signal recognition: Heuristically identifies step or square signal type for 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 or SQUARE)
Polarity	Signal polarity (+1 for positive, -1 for negative pulses)
Amplitude	Absolute difference between the baseline and plateau means; the analysis requires usable baseline and plateau regions
Offset	Mean of the initial baseline, with its original sign in Sigima 1.4.1 or later
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 table and as visual annotations on the signal plot. For step signals, fall-related parameters (fall_time, fwhm) are not applicable; columns containing only unavailable values may be omitted. The feature works best with well-defined pulse signals that have clear baseline regions. For smooth peaks (e.g. Gaussian ones), the amplitude is underestimated and the FWHM overestimated by several percent, depending on the baseline level: for a peak of known shape, prefer a fitting method.

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).

Delete results

Delete analysis results of the selected signals: this submenu lists the results (one entry per result, to delete it individually), followed by a "Delete all results…" entry.

3.2.8 View options for Signals

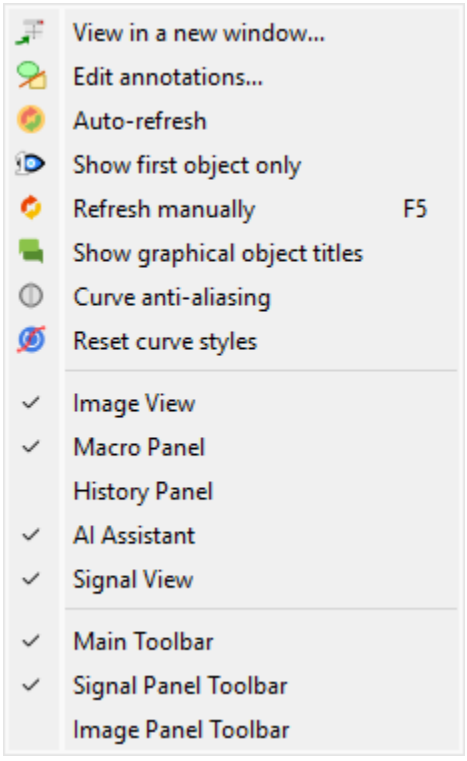


Fig. 37: Screenshot of the "View" menu.

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 stored with the signal (separately from its metadata), so they are persistent and will be displayed every time you visualize the signal.

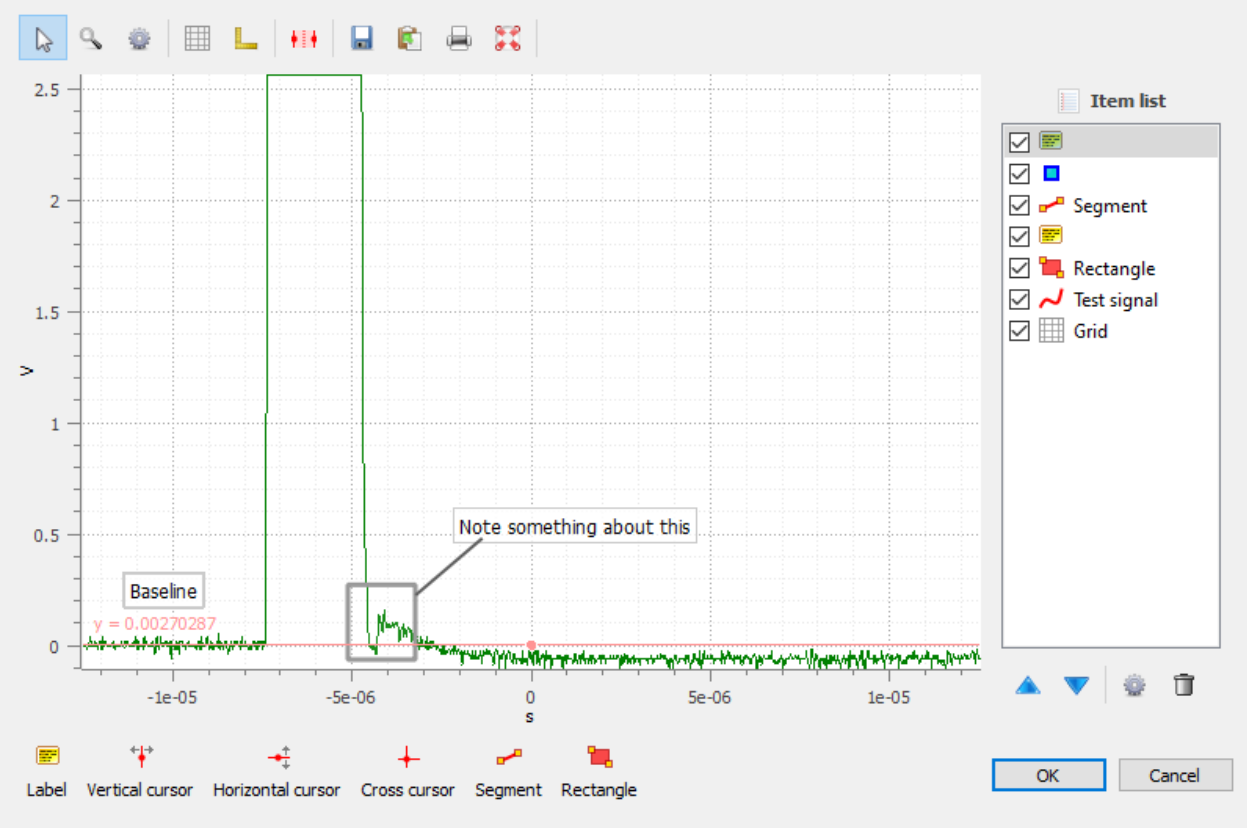


Fig. 38: Annotations may be added in the separate view.

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 with the signal and will be displayed in the main window.

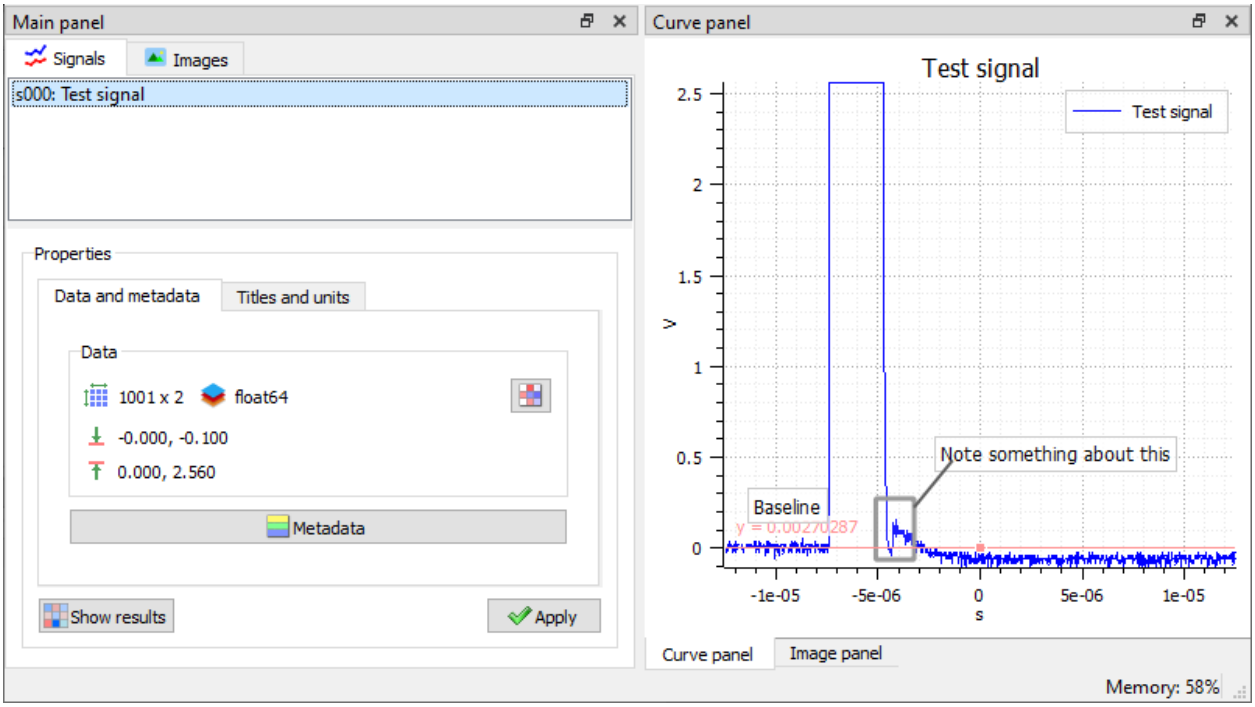


Fig. 39: Annotations are now displayed in the main window.

> **Note**
>
> Annotations may be copied from a signal to another by using the “Copy annotations” and “Paste annotations” actions of the “Edit Annotations” submenu (see Manipulate metadata and annotations).

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 first object only

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

Give the selected signals automatic curve styles again.

When plotting curves, DataLab automatically assigns a color and a line style to each curve, according to its position among the curves shown on the plot (in the order of the signals in the panel): the first curves are drawn with solid lines in distinct colors, readable on both light and dark backgrounds, and the next ones with dashed lines in the same colors. The first curve shown is thus always assigned the first color, whatever the signals shown.

Automatic styles are not saved with the signals. If you choose another color or line style for a curve (with the “Parameters” option of its context menu), this style is saved in the metadata of the signal and is kept afterwards.

This menu entry removes the colors and line styles chosen for the selected signals, so that their curves are assigned automatic styles again.

> **Note**
>
> Workspaces and history files saved with DataLab 1.3 or earlier store the automatic style of each curve: when opening them, these styles are removed from the metadata of the signals, whose curves are assigned the current automatic styles. A color and a line style chosen at that time cannot be told apart from an automatic style when they are identical to one of them: they are removed as well. Files saved with later versions, as well as metadata imported from a file, are not converted.

Panels and toolbars

The last entries of the "View" menu show or hide the panels of the main window: the "Welcome" page (see Welcome page), the "Macro Panel" (see Macros), the "History Panel" (see History Panel), the "AI Assistant" (see AI Assistant), the "Applications" panel (see Applications panel), and the "Signal View" and "Image View" panels. They also show or hide the toolbars of the main window ("Main Toolbar", "Signal Panel Toolbar" and "Image Panel Toolbar").

3.3 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.

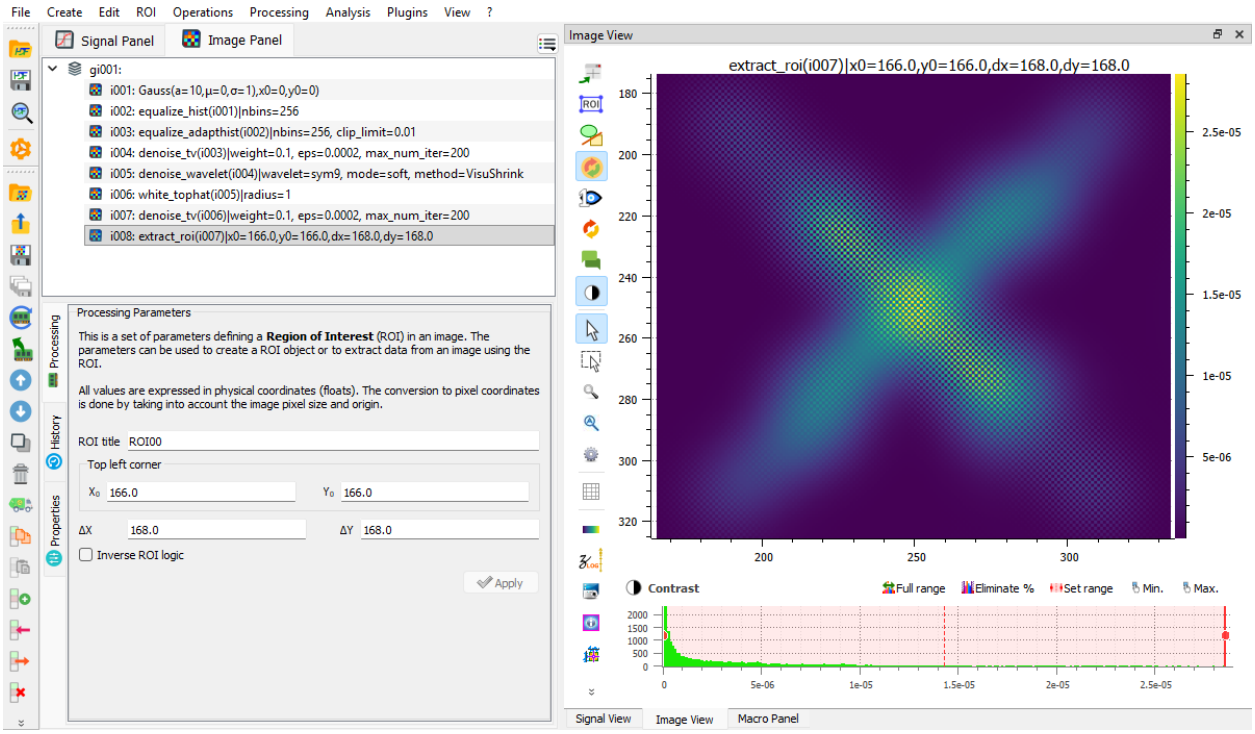


Fig. 40: DataLab main window: Image processing view

3.3.1 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).
- Export, inspect, replay and compare workspace capsules, i.e. a workspace with the provenance of its processing (see Workspace capsules).
- Edit DataLab preferences (see Settings).
- Start or stop the Web API server, copy its connection information, and check its status (see Web API).
- Quit DataLab.

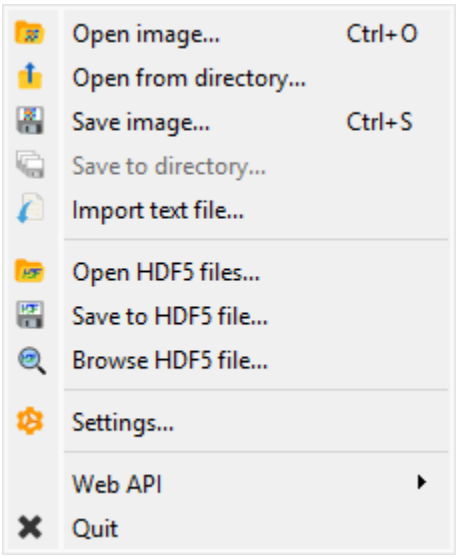


Fig. 41: Screenshot of the "File" menu.

Open image

Create a new image from the following supported filetypes:

File type	Extensions
PNG files	.png
TIFF files	.tif, .tiff
JPEG files	.jpg, .jpeg
JPEG 2000 files	.jp2
Bitmap images	.bmp
NumPy arrays	.npy
MAT-Files	.mat
Text files	.txt, .csv, .asc
HDF5 image files	.h5ima
Andor SIF files	.sif
Princeton Instruments SPE files	.spe
Opticks GEL files	.gel
Hamamatsu NDPI files	.ndpi
PCO Camera REC files	.rec
SPIRICON files	.scor-data
Thales Pixium FXD files	.fxd
Dürr NDT XYZ files	.xyz
FT-Lab files	.ima
Headerless RAW files	.raw

> **Note**
>
> Thales Pixium FXD files, and the writing of Dürr NDT XYZ files, are supported by the built-in image formats plugin (`datalab_imageformats`).

When opening a headerless RAW file (`.raw`), DataLab displays a RAW image import dialog to describe its binary layout:

pixel data type, image width and height, offset to the first image, number of images, gap between consecutive images, byte order, and the White is zero option (which displays the imported image with an inverted gray colormap without altering pixel values). Multi-image files create one image per frame. When several RAW files are opened at once, DataLab asks whether to apply the same import parameters to all of them or to set the parameters of each file individually.

> **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 set the `imageio_formats` option of the Sigima library (`sigima.config.options.imageio_formats`), for example from a plugin. This option is formatted like the IMAGEIO_FORMATS object which represents the natively supported formats (formats added this way can only be opened, not saved):
>
> `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 to the following supported filetypes (the other file types listed in "Open image" can only be opened):

File type	Extensions
PNG files	.png
TIFF files	.tif, .tiff
JPEG files	.jpg, .jpeg
JPEG 2000 files	.jp2
Bitmap images	.bmp
NumPy arrays	.npy
MAT-Files	.mat
Text files	.txt, .csv, .asc
HDF5 image files	.h5ima
Dürr NDT XYZ files	.xyz

After choosing the destination file, DataLab opens an Export image dialog tailored to the selected file format, with a live preview of the exported image (displayed in gray levels as it will be read back from the file):

- Data preparation: normalization mode (none, min-max, percentile or manual bounds), out-of-range behavior (clip or rescale), non-finite value policy (error, clip or replace), gamma correction, inversion and target data type among those supported by the destination format. When a format only supports integer pixels (e.g. PNG, JPEG or BMP) and the source image uses another data type, min-max rescaling is selected by default.

- Format-specific options: compression, quality, predictor, tiling, resolution, photometric interpretation, text delimiter and precision, or MAT compression, depending on the chosen format.

Save images to directory

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

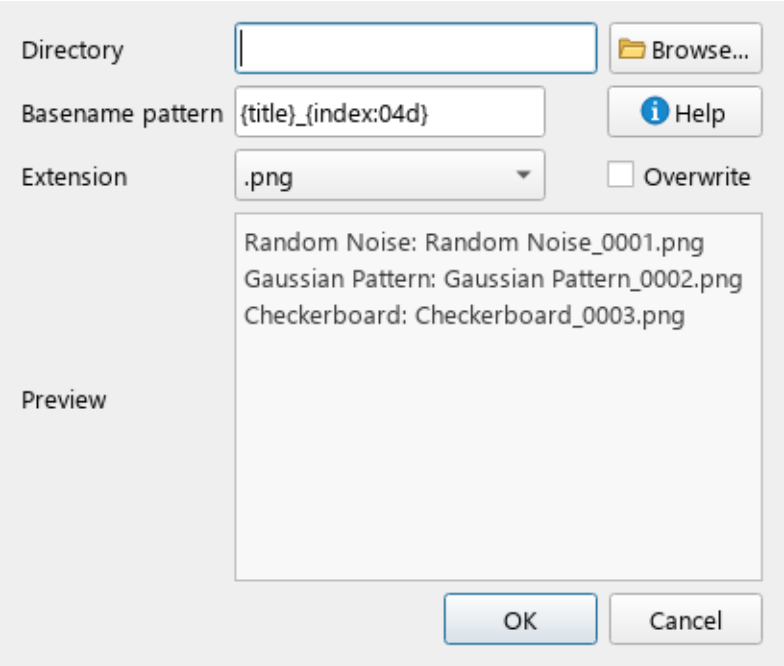


Fig. 42: Save images to directory dialog.

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
- Basename pattern: Define a pattern for the filenames using Python format strings (the Help button describes the available placeholders)
- 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 (each file name is preceded by the group and the title of the corresponding image)

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}`: 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.

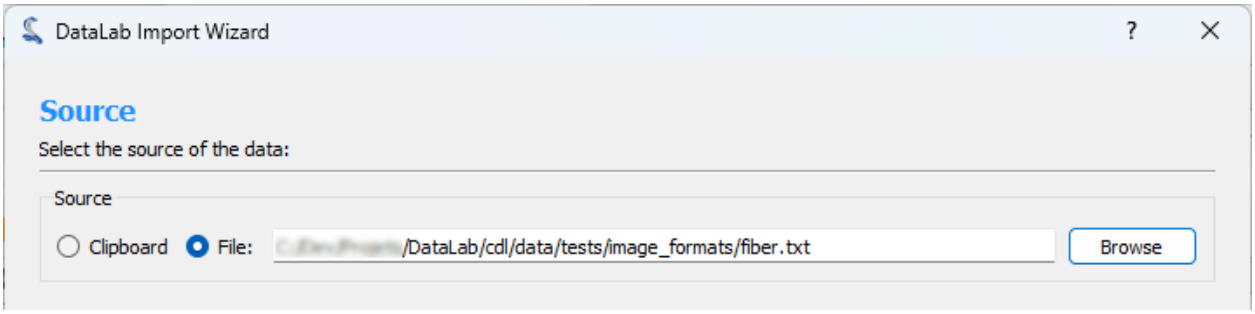


Fig. 43: Step 1: Select the source

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.
- Decimal: The decimal separator (point or comma).
- 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.
- Max. nb of rows: The maximum number of rows to import. If the file contains more rows, they will be ignored.
- Header: The row containing the column names, if any (none, inferred from the data, or first row).
- 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.

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.

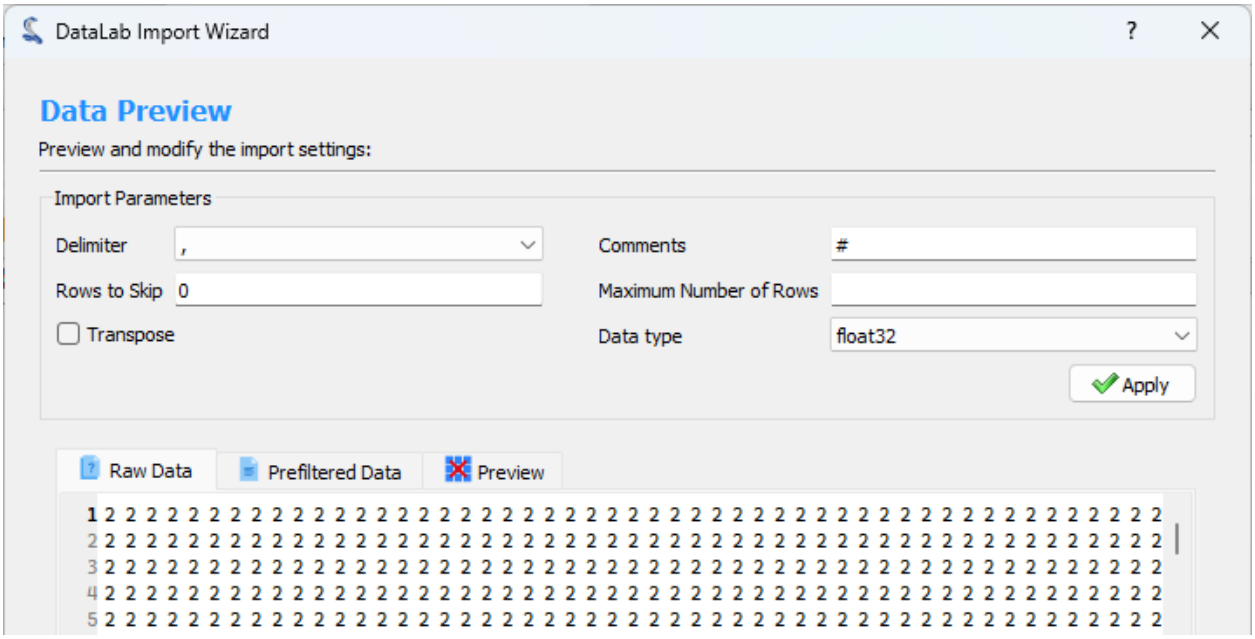


Fig. 44: Step 2: Configure the import

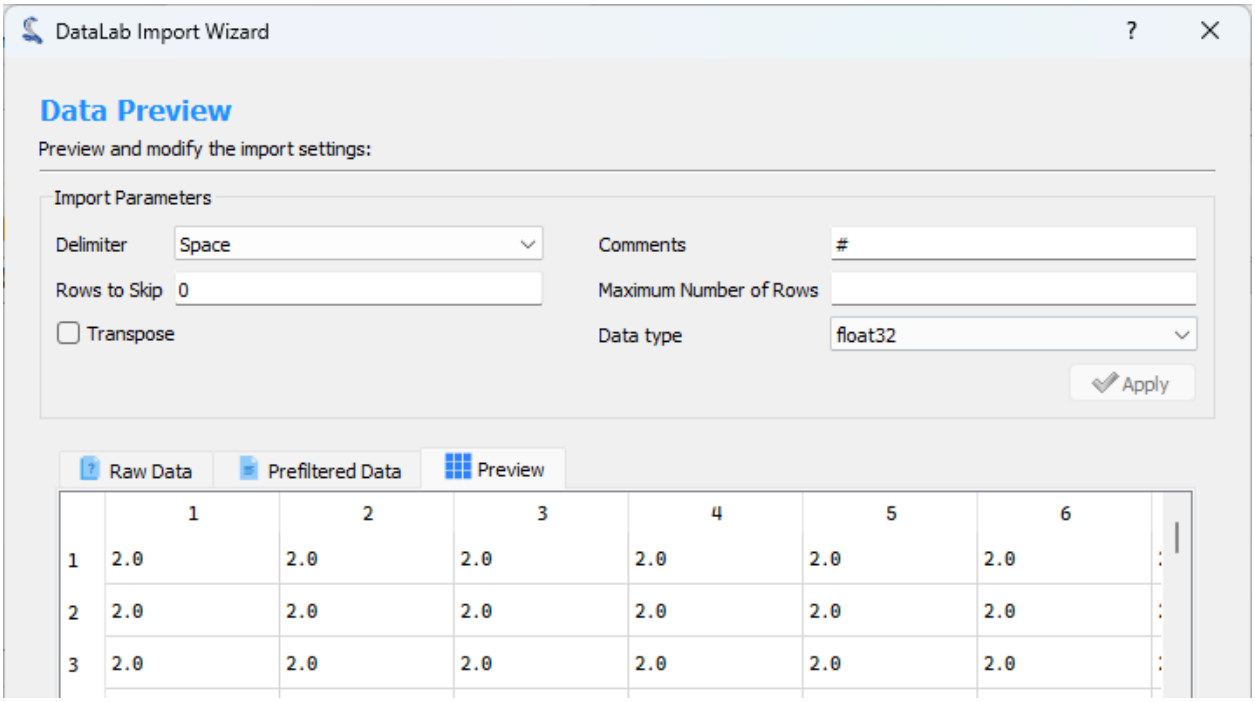


Fig. 45: Step 2: Preview the result

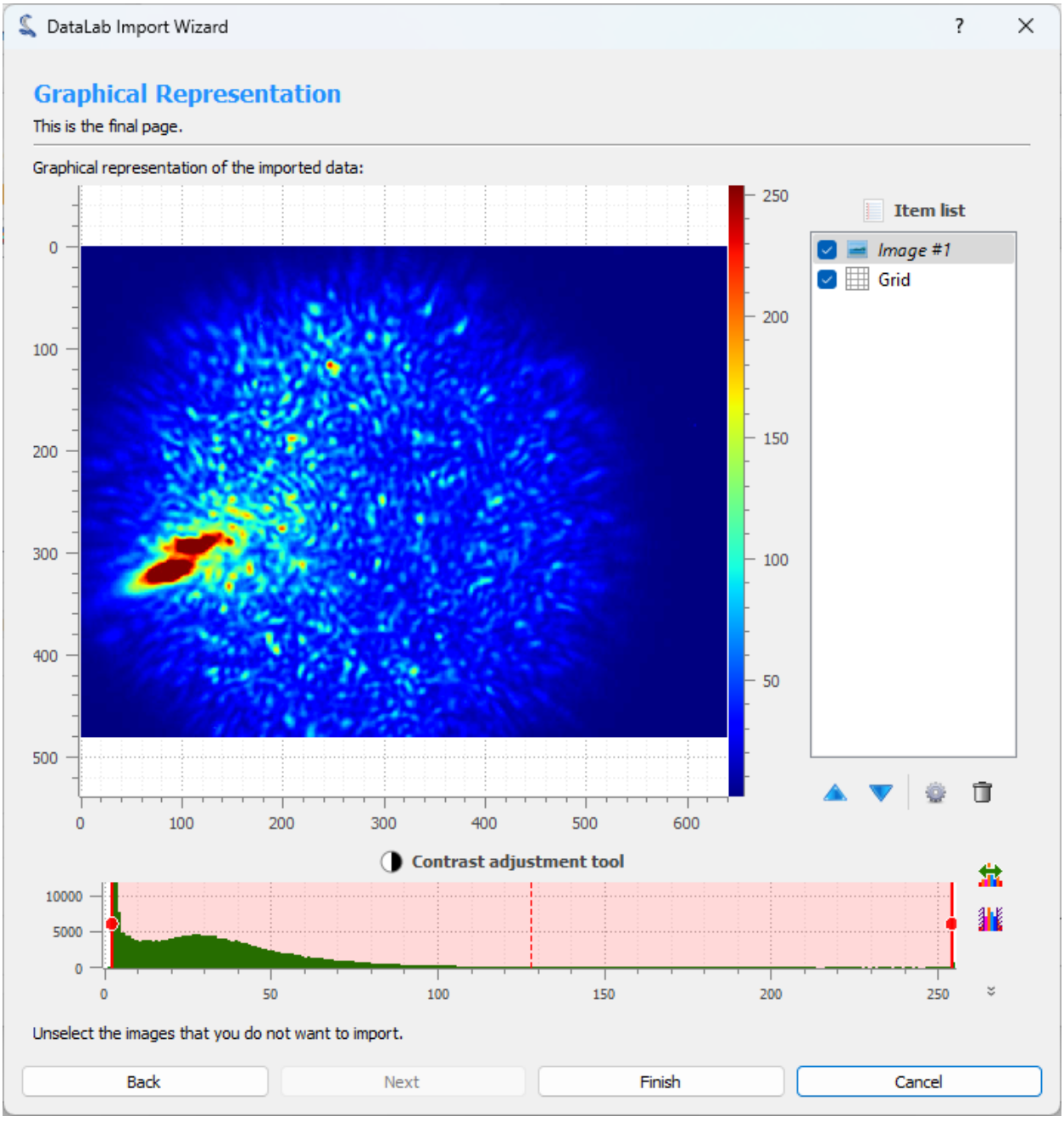


Fig. 46: Step 3: Show graphical representation

3.3.2 Create Images

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

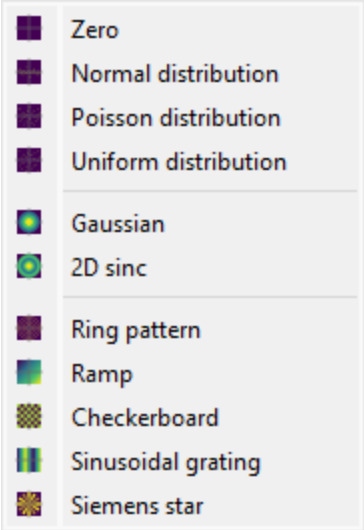


Fig. 47: Screenshot of the “Create” menu.

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
	Gaussian	$z = A \cdot \exp\left(-\frac{\left(\sqrt{(x-x_0)^2+(y-y_0)^2}-\mu\right)^2}{2\sigma^2}\right)$
	Ramp	$z = A(x-x_0) + B(y-y_0) + C$
	Checkerboard	Alternating square pattern for calibration and spatial frequency analysis
	Sinusoidal grating	$z = A\sin(2\pi(f_x \cdot x + f_y \cdot y) + \varphi) + C$

continues on next page

Table 3 – continued from previous page

Icon	Model	Equation
	Ring pattern	Concentric circular rings for radial analysis
	Siemens star	Radial spoke pattern for resolution testing
	2D sinc	$z = A \cdot \mathrm{sinc}\left(\frac{\sqrt{(x-x_0)^2+(y-y_0)^2}}{\sigma}\right) + C$

The 2D sinc uses the unnormalized definition $\mathrm{sinc}(u) = \frac{\sin(u)}{u}$, with $\mathrm{sinc}(0) = 1$, as the cardinal sine signal: it first vanishes at the distance $\pi\sigma$ from its center.

The Gaussian, ramp, sinusoidal grating, ring pattern, Siemens star and 2D sinc models are computed over a domain $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. When the Physical coordinates option is enabled (default), this domain defines the image coordinates: the axes show x and y as used in the equation, so that centers, widths and spatial frequencies can be read directly on the image (e.g. a sinusoidal grating of frequency $f_x = 0.1$ shows one period every 10 units along the X axis). When the option is disabled, the image is displayed in pixel coordinates.

> **Note**
>
> Images created with DataLab 1.3 or earlier were displayed in pixel coordinates: they keep these coordinates when their creation parameters are edited or recomputed.

3.3.3 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”.
- Manipulate metadata and annotations associated with the current image, thanks to the “Metadata” and “Annotations” submenus which provide the following features.

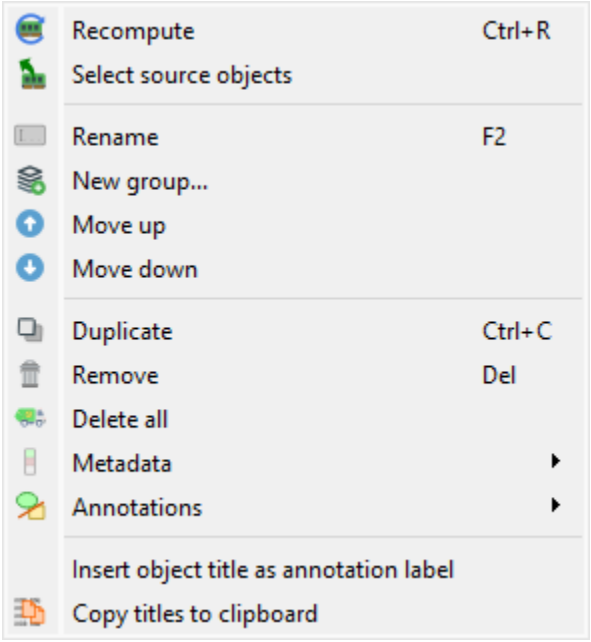


Fig. 48: Screenshot of the "Edit" menu.

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.

It also refreshes analysis results (statistics, FWHM, centroid, peak/contour/blob detection, etc.) using their stored parameters. Analysis results are not recomputed automatically when you modify a region of interest, the data, or object properties: existing results are left untouched until you explicitly trigger this action, giving you full control over when analyses are refreshed.

> **Note**
>
> This action is enabled as soon as an image or a group is selected, but it only applies to images that have stored processing parameters (created through processing operations) or stored analysis parameters (analysis operations previously run on them). When none of the selected images has such parameters, an information message is displayed.

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 enabled when exactly one image is selected. When that image has no source object references, an information message is displayed.

Metadata

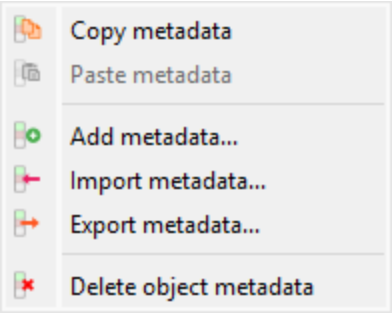


Fig. 49: Screenshot of the "Metadata" submenu.

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.

When pasting metadata, a dialog box allows you to select which metadata categories to keep from the clipboard. Annotations are copied and pasted separately:

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 object metadata

When deleting metadata using the "Delete object 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.

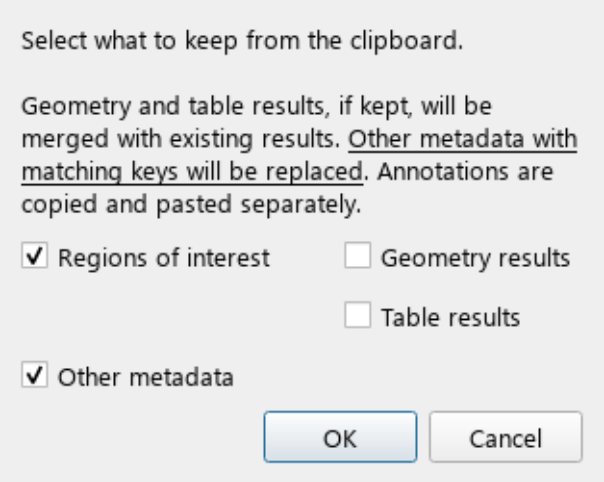


Fig. 50: Screenshot of the "Paste metadata" dialog.

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.

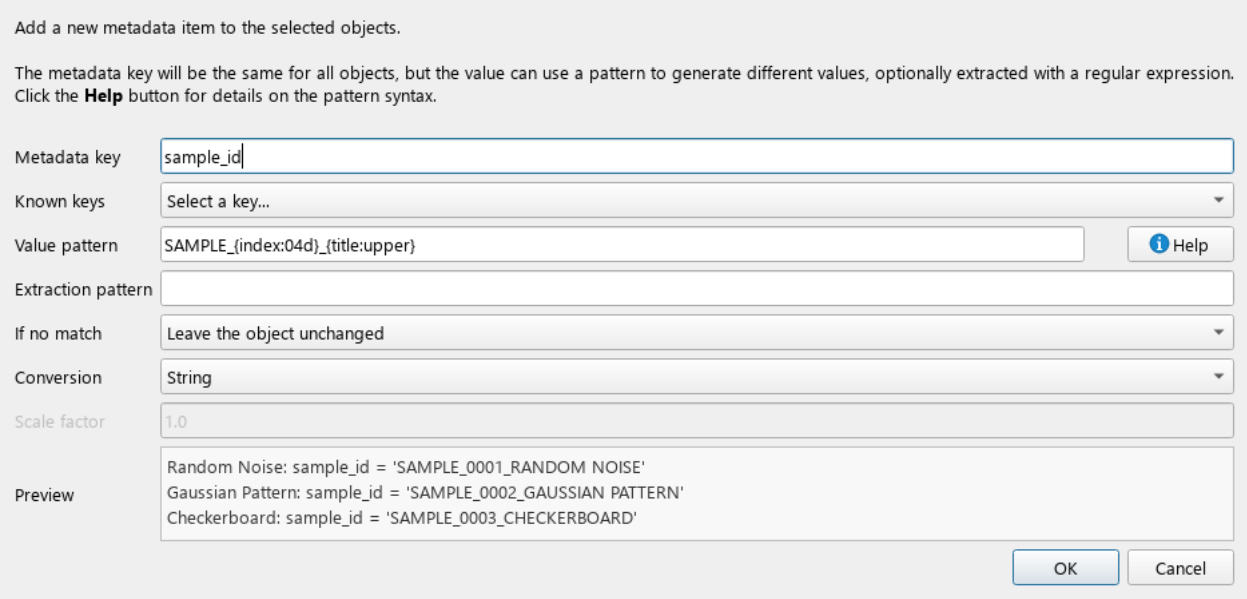


Fig. 51: Add metadata dialog.

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 (letters, digits, _, . and -, e.g. a plugin key such as `plugin.org.example.my-plugin.gain`)
- Known keys: Pick a key already present on the selected images, or a key expected by the methods of the installed application plugins, to copy it into the metadata key
- Value pattern: Define a pattern for the metadata value using Python format strings

- Extraction pattern: Optionally, extract the value from the formatted text with a Python regular expression: its first group, or the whole match if it has no group, becomes the value
- If no match: Leave images without a match unchanged (default), or report an error
- Conversion: Choose how to store the value (string, float, integer, or boolean)
- Scale factor: Multiply numeric values, e.g. `0.001` to convert milliseconds to seconds
- Preview: See how the metadata will be added to each selected image, and which images are left unchanged

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
- Read exposure times from titles: key=``exposure_time_s``, pattern=``{title}``, extraction=``([d.]+)s*ms``, conversion=float, scale factor=``0.001`` → Sets `exposure_time_s=0.005` on an image titled “Flat 5 ms 01” and leaves “Dark 01” unchanged

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.

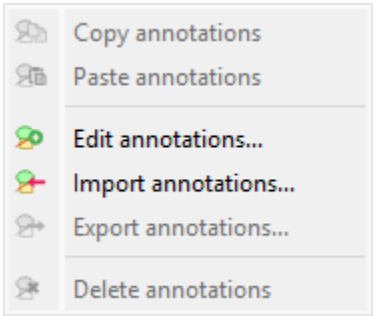


Fig. 52: Screenshot of the “Annotations” submenu.

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:

- "Insert object title as annotation label": 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 (titles of processed images refer to their sources by the first 8 characters of their unique identifier, prefixed by "g" for groups):

```
Raw images
    Gauss(a=10,μ=0,σ=1,x0=0,y0=0)
    moving_median(9f53944b)|size=5, mode=nearest
    fft(76e4ce1b)
log10(g9f72bd95)
    log10(9f53944b)
    log10(76e4ce1b)
    log10(1462f722)
```

3.3.4 Regions Of Interest (ROI)

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

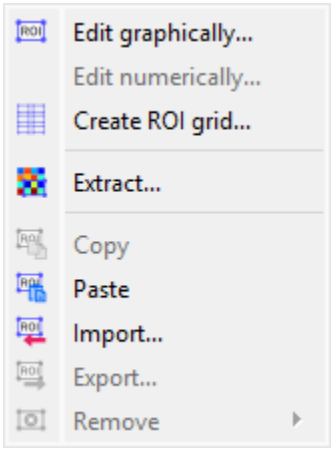


Fig. 53: Screenshot of the "ROI" menu.

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.
- The "Processing" menu actions are performed 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, or 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 generally analyze each ROI if defined. Statistics also include a whole-image row, identified by ROI index `-1`.

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

The “ROI” menu allows you to:

- “Edit graphically” ROI: 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).

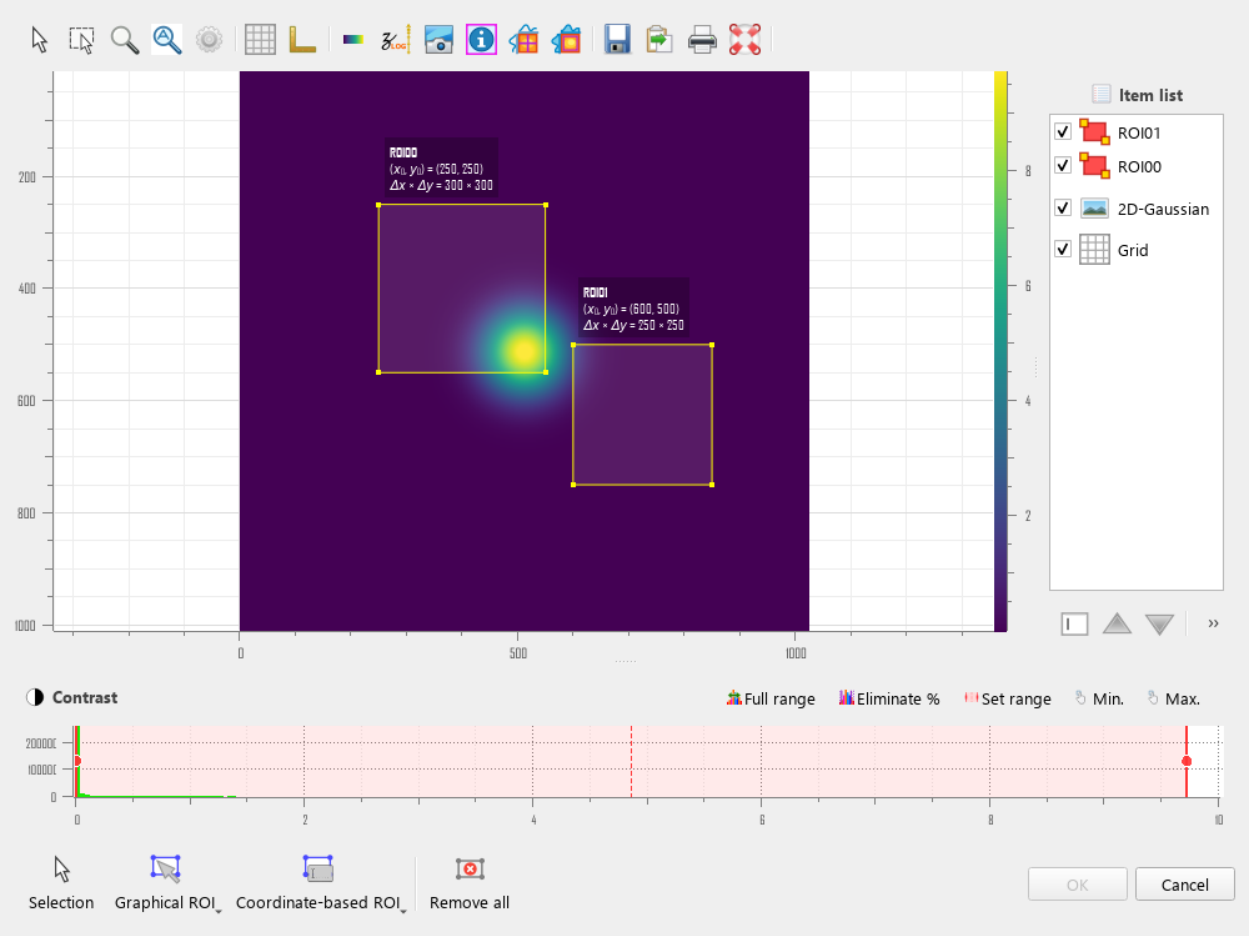


Fig. 54: An image with an ROI.

- “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" : remove a single ROI (one entry per ROI of the selected image) or all defined ROIs for the selected images ("Remove all").

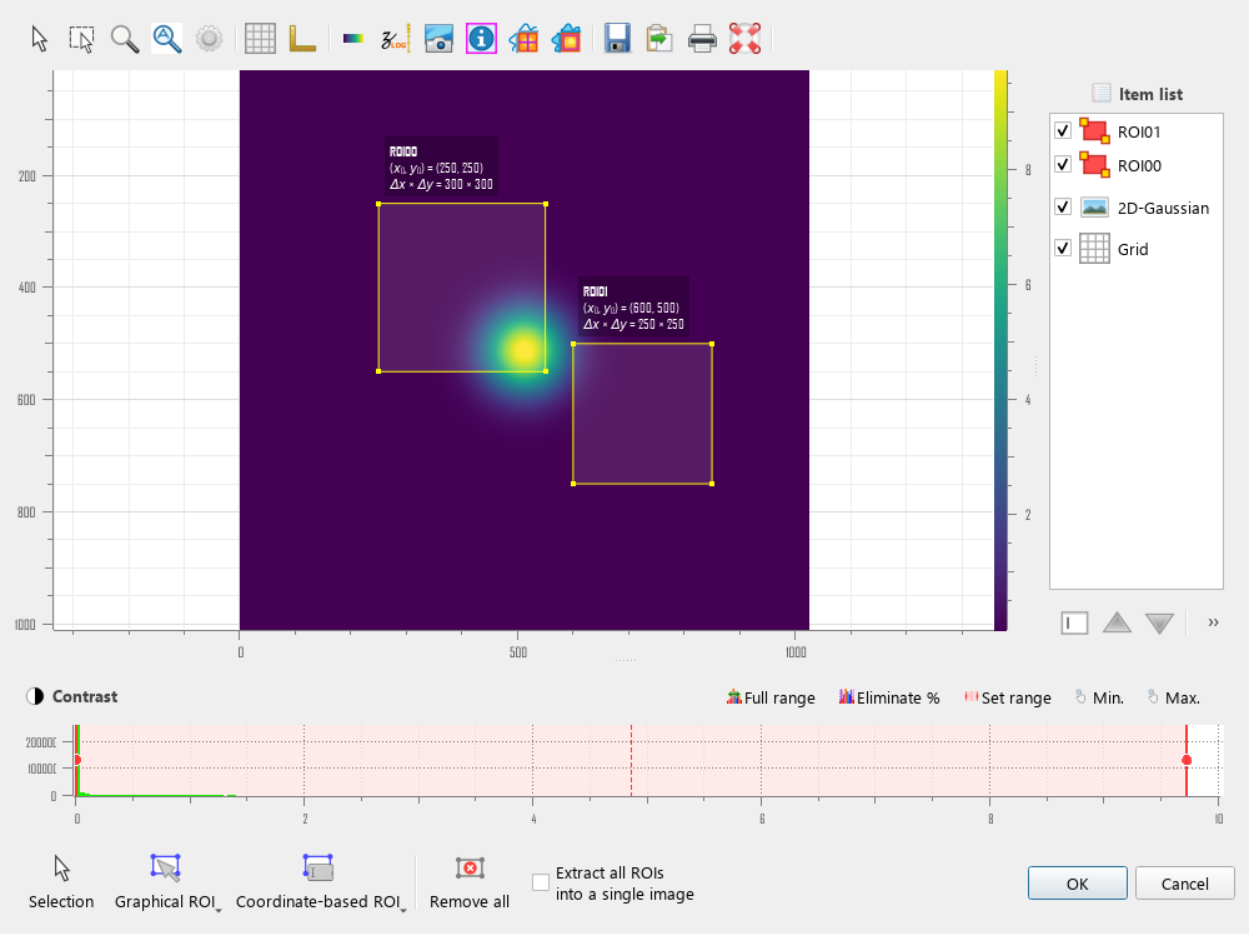


Fig. 55: ROI extraction dialog: the ROI is defined by moving the position and adjusting the size of a rectangle shape.

3.3.5 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.

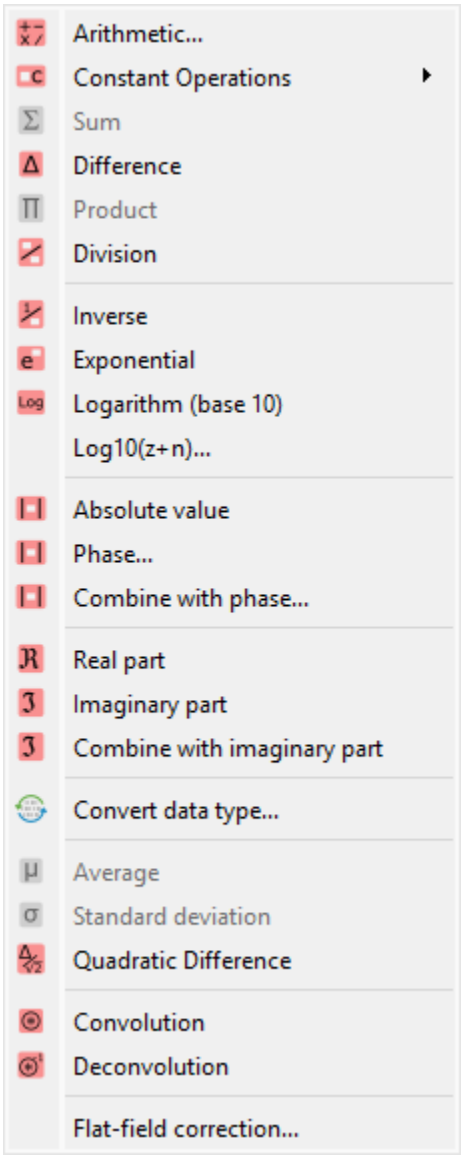


Fig. 56: Screenshot of the “Operations” menu.

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, such as arithmetic operations, mathematical functions, convolution, or flat-field correction.

Arithmetic

The "Arithmetic" action creates a new image which is the result of an arithmetic operation between each selected image and a second image (selected in a dialog box):

$$z = (z_1 \odot z_2) \times a + b$$

where z_1 is the selected image, z_2 is the second image, $\odot$ is the Operator ($+$, $-$, $\times$ or $/$), a is the Factor and b is the Constant. The Operation field shows the resulting expression. By default, the result is converted to the data type of the selected image.

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+n)	$z_k = \log_{10}(z_k + n)$ (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
μ Average	$z_M = \frac{1}{M}\sum_{k=0}^{M-1} z_k$
σ Standard Deviation	$z_M = \sqrt{\frac{1}{M}\sum_{k=0}^{M-1} (y_k - \bar{y})^2}$
Δ/√2 Quadratic difference	$z_2 = \frac{z_1 - z_0}{\sqrt{2}}$

Flat-field correction

Apply flat-field correction to each selected raw image, using a separately selected flat-field image or a matching flat-field image for each source:

$$z_1 = \begin{cases} \frac{z_0}{z_f}.\overline{z_f} & \text{if } z_0 > z_{threshold} \text{ and the pixel is in the ROI} \\ z_0 & \text{otherwise} \end{cases}$$

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 average over the entire flat-field image, ignoring NaNs. For an image with no NaNs:

$$\overline{z_f} = \frac{1}{N_{row}.N_{col}}. \sum_{i=0}^{N_{row}-1} \sum_{j=0}^{N_{col}-1} z_f(i,j)$$

If no ROI is defined, all pixels above the threshold are corrected. With Sigima 1.4.1 or later, a zero flat-field value at a corrected pixel is rejected; zero values outside the ROI or at pixels excluded by the threshold are allowed. Flat-field values close to zero strongly amplify the noise of the corrected pixels. With Sigima 1.4.1 or later, correcting an integer image gives a floating-point image, instead of wrapping the values exceeding the range of the image type.

> **Note**
>
> Raw image and flat field image are supposedly already corrected by performing a dark frame subtraction.

3.3.6 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.

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.

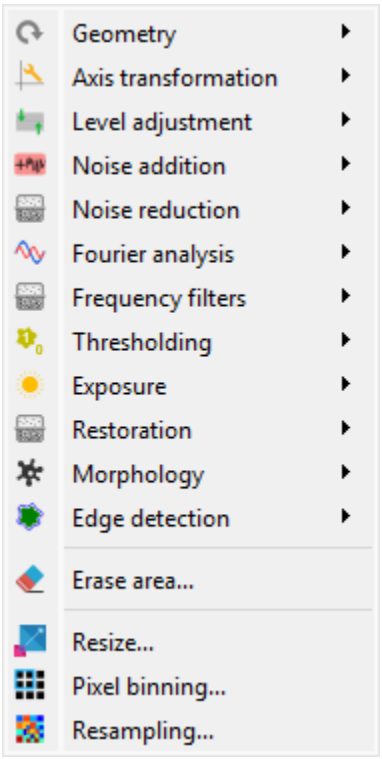


Fig. 57: Screenshot of the "Processing" menu.

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 (up to degree 3) of each selected image with respect to the X, Y or Z axis (Calibrate parameter, Z axis by default):

$$t' = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

where t stands for the X, Y or Z values of the image.

> **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	$z_1 = \frac{z_0 - z_{\min}}{z_{\max} - z_{\min}}$
Area	$z_1 = \frac{z_0}{\sum_{i=0}^{N-1} z_i}$
Energy	$z_1 = \frac{z_0}{\sqrt{\sum_{n=0}^{N-1} \lvert z_0[n] \rvert^2}}$
RMS	$z_1 = \frac{z_0}{\sqrt{\frac{1}{N} \sum_{n=0}^{N-1} \lvert z_0[n] \rvert^2}}$

Here N is the number of pixels, indexed over the flattened image, assuming no NaNs. The implemented sum and mean ignore NaNs.

Clipping

Apply the clipping to each selected image.

Replace special values

Create a new image where `NaN`, `+Inf` and `-Inf` pixels are replaced using a configurable strategy per target value.

Each of the three targets (`NaN`, `+Inf`, `-Inf`) is handled independently. For each target the user selects one of the following strategies (or Do nothing to leave the target untouched):

Strategy	Description
Do nothing	Leave the target values untouched.
Replace with zero	Replace by `0`.
Replace with constant	Replace by a user-defined constant value.
Replace with minimum / maximum	Replace by the minimum / maximum of the finite pixels.
Replace with mean / median	Replace by the mean / median of the finite pixels.
N-neighbor minimum / maximum / mean / median	Replace by the corresponding statistic computed on the `(2N+1) × (2N+1)` neighborhood around each affected pixel.

Before opening the standard parameter form, the dialog displays a count of each target found in the image (e.g. NaN: 124 (0.5 %)). When a neighbor strategy is selected, a small preview shows the 2D kernel that will be used.

> **Note**
>
> Unlike for signals, no interpolation strategy is currently available for images.

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' or 'Around' (zeros distributed around the image)

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)	$z_1 = \lvert\mathrm{FFT}(z_0)\rvert$ or $z_1 = 20\log_{10}(\lvert\mathrm{FFT}(z_0)\rvert)$ (dB)
Phase spectrum	Phase of the FFT expressed in degrees, using numpy.angle	$z_1 = \angle(\mathrm{FFT}(z_0))$
Power spectral density	Optional: output in decibels (dB)	$z_1 = \lvert\mathrm{FFT}(z_0)\rvert^2$ or $z_1 = 10\log_{10}\left(\lvert\mathrm{FFT}(z_0)\rvert^2\right)$ (dB)

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

Frequency filters

The following frequency-domain filters are available:

Method	Description
Butterworth	Butterworth filter, based on skimage.filters.butterworth
Gaussian bandpass	Gaussian band-pass filter applied in the frequency domain, centered on the Center frequency f_0 with a standard deviation **σ** (both in cycles per pixel): the FFT of the image is multiplied by $\exp\left(-\frac{1}{2}\left(\frac{f-f_0}{\sigma}\right)^2\right)$, where f is the radial frequency. Depending on the Inverse FFT result parameter, the real part (Real part, the default) or the modulus (Absolute value) of the inverse FFT is returned: for a real image, the modulus is the absolute value of the real part.

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 new images by applying exposure corrections. Brightness and contrast uses the first selected image to initialize one shared input window, then applies that window independently to every selected image. Each result retains the output range of its own source image.

The following functions are available:

Function	Implementation	Comments
Brightness and contrast	`sigima.proc.image.adjust_brightness_cor`	Histogram-driven clipped linear remapping, initialized from the first selected image
Gamma correction	skimage.exposure.adjust_gamma	
Logarithmic correction	skimage.exposure.adjust_log	
Sigmoid correction	skimage.exposure.adjust_sigmoid	
Histogram equalization	skimage.exposure.equalize_hist	
Adaptive histogram equalization	skimage.exposure.equalize_adapt	Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm
Intensity rescaling	skimage.exposure.rescale_intensi	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_c	
Bilateral filter denoising	skimage.restoration.denoise_bilat	
Wavelet denoising	skimage.restoration.denoise_wav	
White Top-Hat denoising	skimage.morphology.white_topha	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.

Edge detection

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 edge detection 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
x_{min}	X-coordinate of the center of the first output pixel
x_{max}	X-coordinate of the center of the last output pixel
y_{min}	Y-coordinate of the center of the first output pixel
y_{max}	Y-coordinate of the center of the last output pixel
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.
ΔX	Pixel size in x-direction (if 'Pixel size' mode is selected)
ΔY	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)

3.3.7 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.

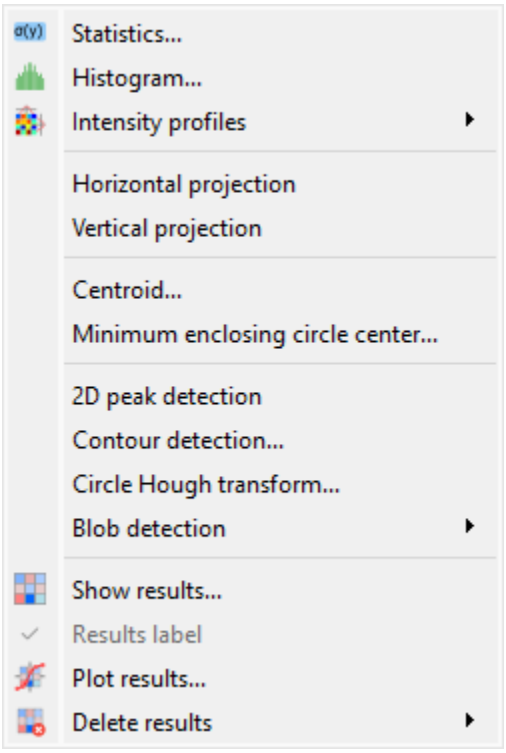


Fig. 58: Screenshot of the "Analysis" menu.

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. The "Histogram", "Intensity profiles" and projection features are exceptions: they create new signals from each selected image.

Statistics

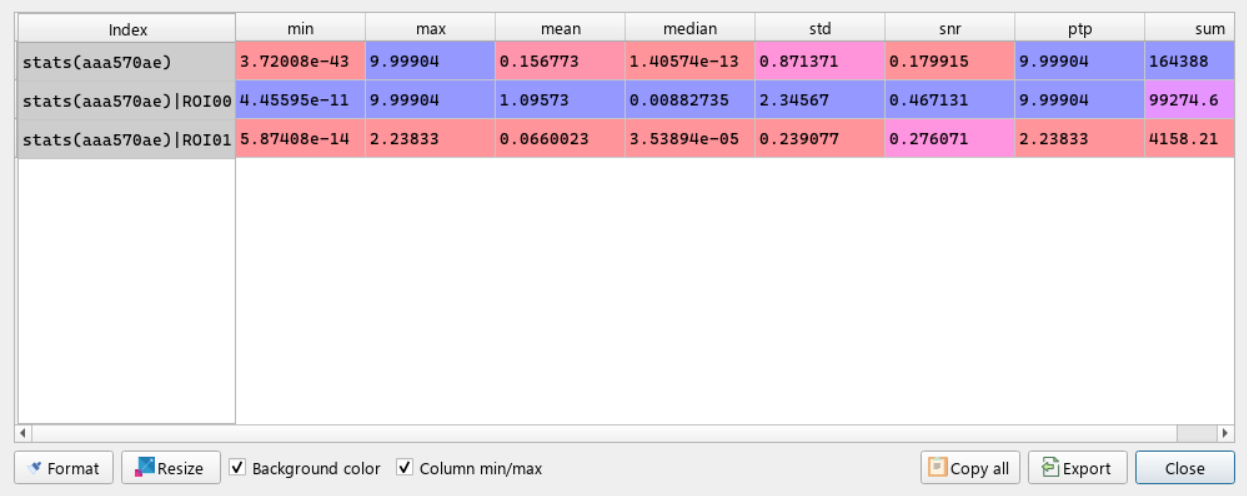
Compute statistics on selected image and show a summary table.

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



Index	min	max	mean	median	std	snr	ptp	sum
stats(aaa570ae)	3.72008e-43	9.99904	0.156773	1.40574e-13	0.871371	0.179915	9.99904	164388
stats(aaa570ae)|ROI00	4.45595e-11	9.99904	1.09573	0.00882735	2.34567	0.467131	9.99904	99274.6
stats(aaa570ae)|ROI01	5.87408e-14	2.23833	0.0660023	3.53894e-05	0.239077	0.276071	2.23833	4158.21

Format | Resize | Background color | Column min/max | Copy all | Export | Close

Fig. 59: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

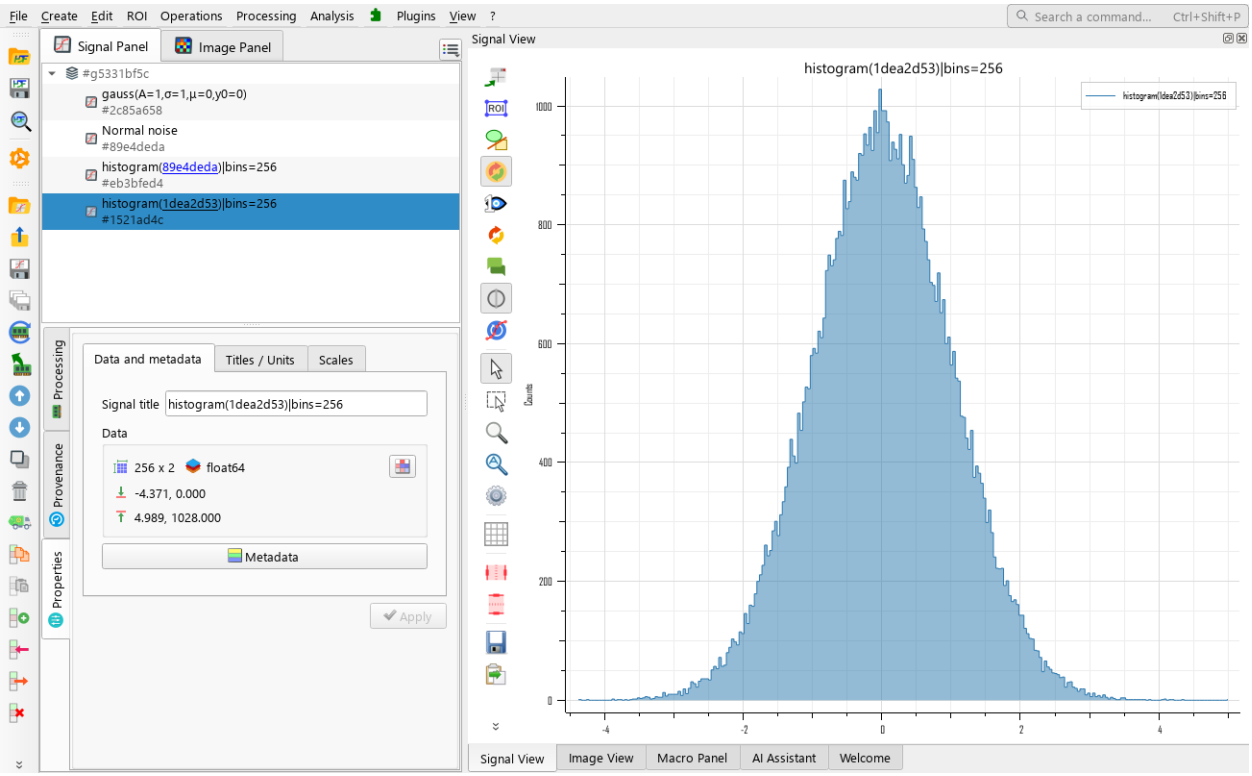


Fig. 60: Example of histogram.

Intensity profiles

Profiles are defined in the physical coordinates of the image, like regions of interest and analysis results. The X axis of the resulting signals holds physical coordinates (line and average profiles) or distances (segment and radial profiles), with the unit of the corresponding axis of the image. Segment and radial profiles require the same unit on the X and Y axes of the image.

Line profile

Extract an horizontal or vertical profile from each selected image, and create new signals from these profiles. The profile is extracted along the row (or column) of pixels whose center is the nearest to the given Y (or X) coordinate.

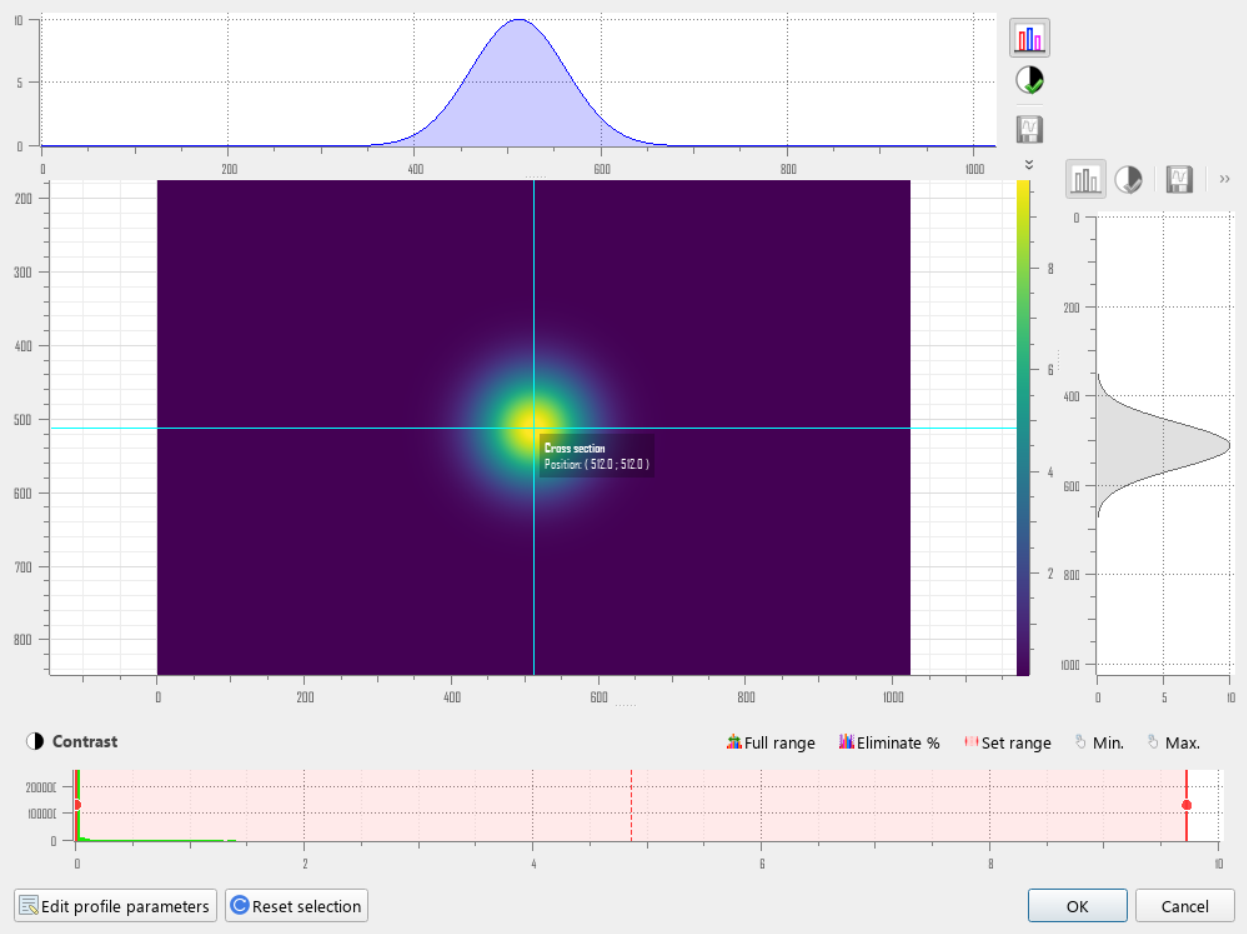


Fig. 61: Line profile dialog. Parameters may also be set manually (“Edit profile parameters” button).

Segment profile

Extract a segment profile from each selected image, and create new signals from these profiles. The segment is sampled at regular steps from its start point to its end point included, with one sample per pixel along the axis where it spans the most pixels, each sample taking the value of the nearest pixel.

Average profile

Extract an horizontal or vertical profile averaged over a rectangular area, from each selected image, and create new signals from these profiles. The pixels whose centers lie inside the rectangle (bounds included) are averaged, as for rectangular regions of interest.

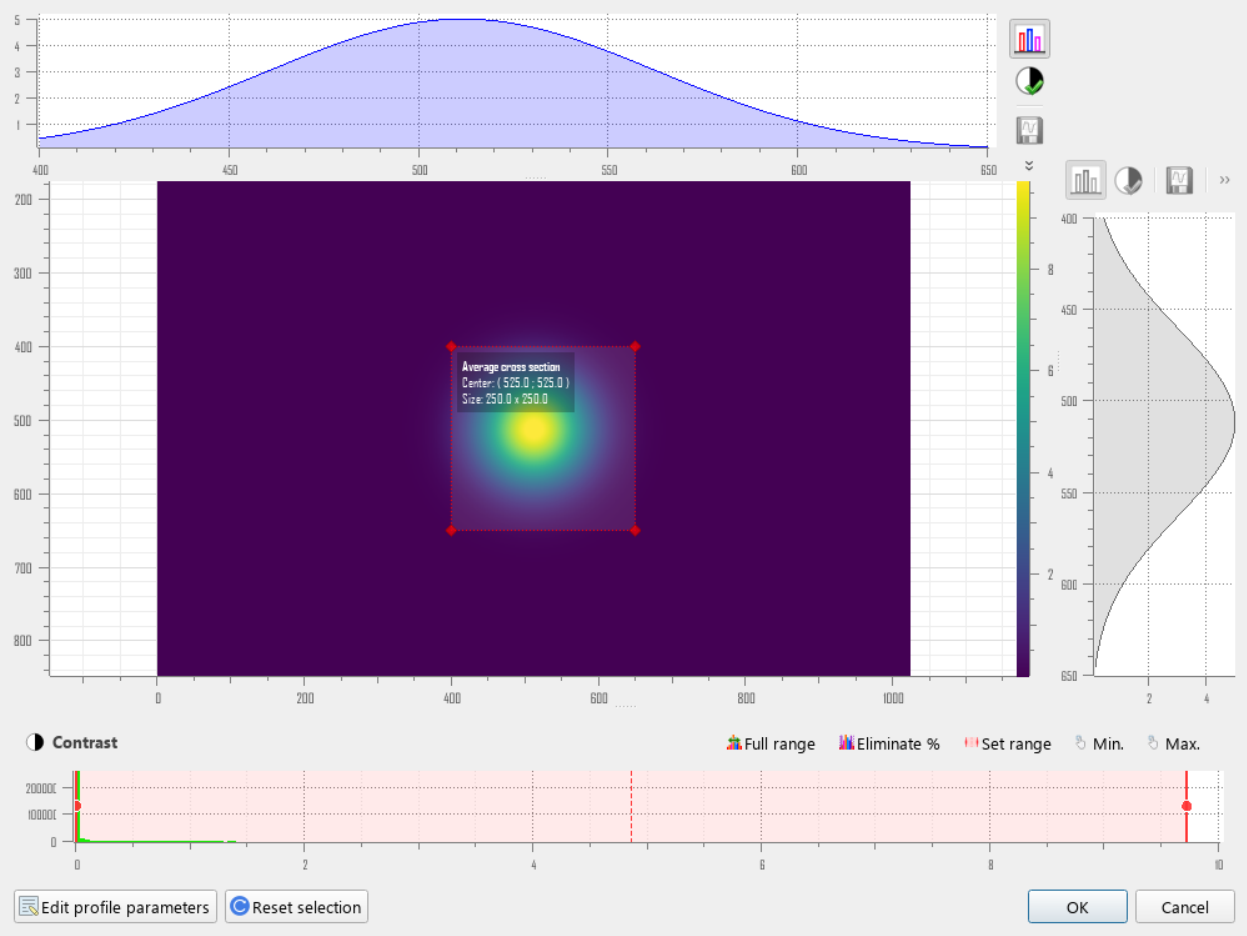


Fig. 62: Average profile dialog: the area is defined by a rectangle shape. Parameters may also be set manually ("Edit profile parameters" button).

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 physical units
Y	Y coordinate of the center (if user-defined), in physical units

Horizontal and vertical projections

Compute horizontal and vertical projection profiles:

- Horizontal projection: Sum the pixel values along each column (projection on the x-axis).
- Vertical projection: Sum the pixel values along each row (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()` computes a reference centroid with a projected profile-based method (`sigima.tools.image.get_projected_profile_centroid()`), and selects, between the Fourier result and a standard centroid from scikit-image, the one which is the most consistent with this reference.

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

With Sigima 1.4.1 or later, compute the smallest circle containing the physical centers of the pixels strictly above the midpoint of the image value range. The X and Y units must match. Earlier versions fit a contour and could leave selected pixel centers outside the circle. A single isolated hot pixel above the threshold enlarges the circle: the result is not a robust measure of a beam width.

2D peak detection

Automatically find peaks on image using a minimum-maximum filter algorithm.

> 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.

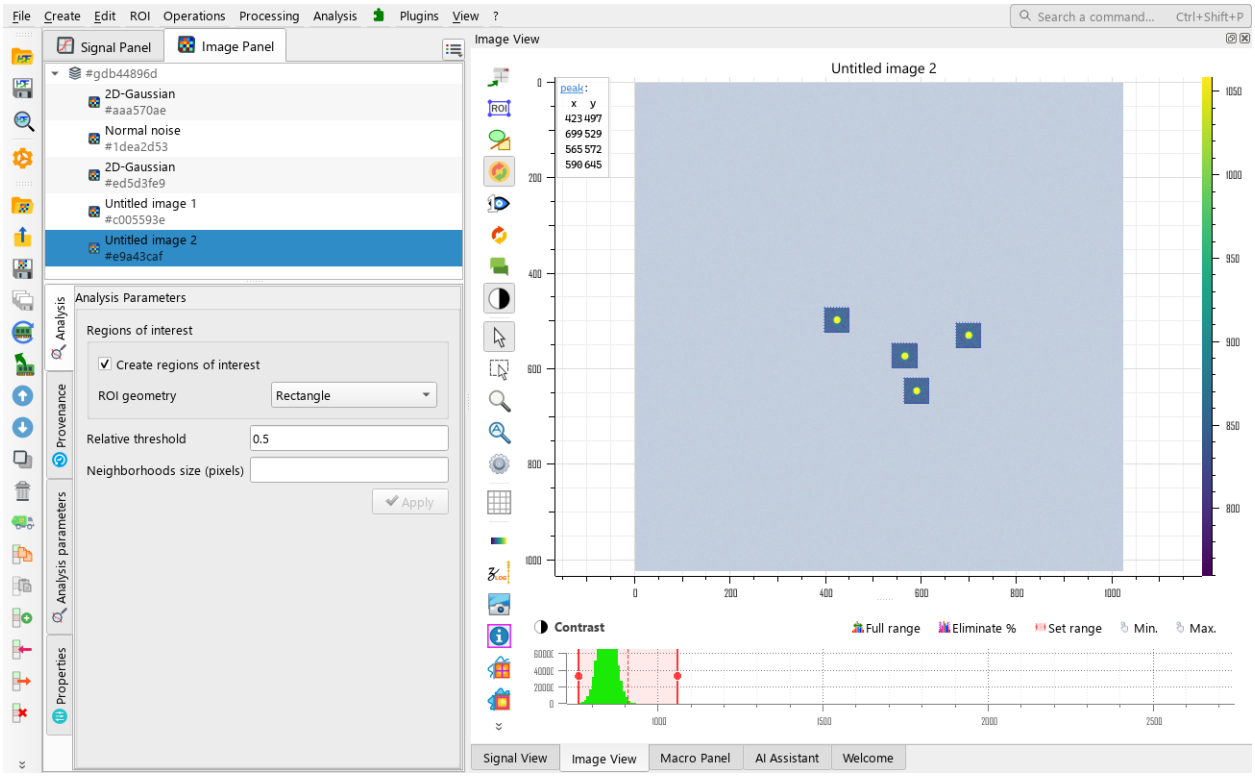


Fig. 63: Example of 2D peak detection.

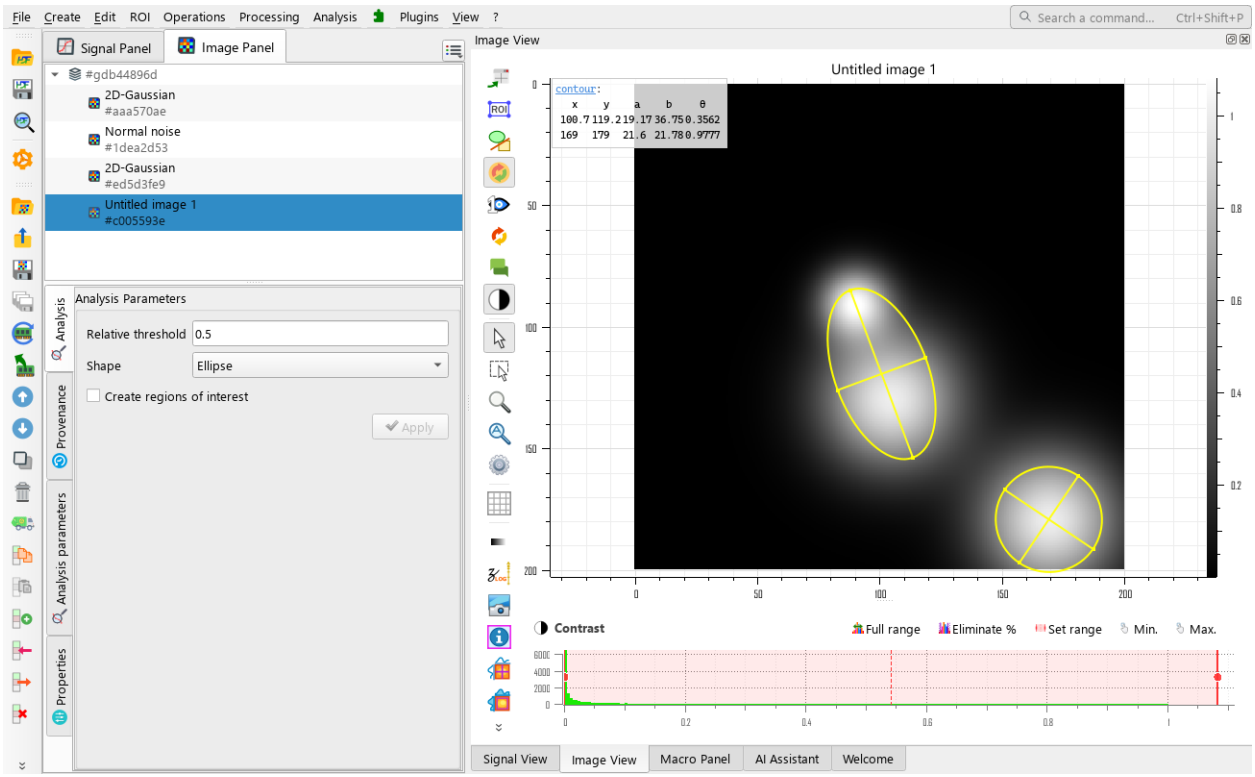


Fig. 64: Example of contour detection.

> **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.

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. Except for contour detection (whose ROIs follow the shape of each detected contour, see Contour Detection), 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.
>
> The created ROIs are added to the regions of interest already defined on the image, instead of replacing them: running several detections in a row accumulates their results, and no destructive confirmation dialog is shown. They are ordinary ROIs, which may be moved, resized, or deleted afterwards like any manually defined region.

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).

Delete results

Delete analysis results of the selected images: this submenu lists the results (one entry per result, to delete it individually), followed by a "Delete all results…" entry.

3.3.8 View options for Images

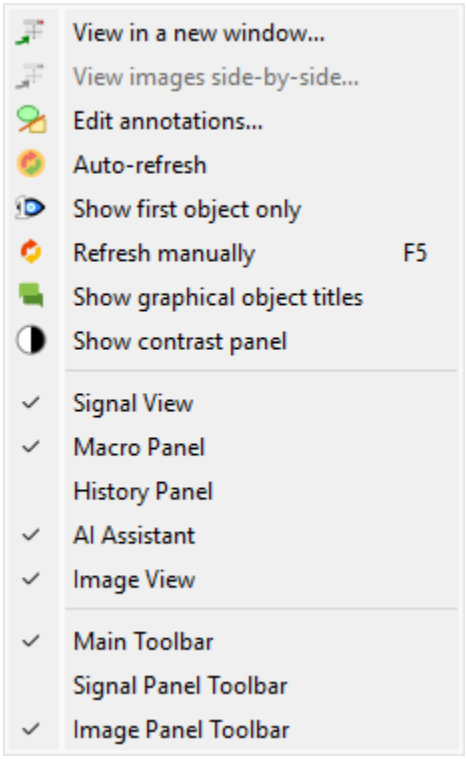


Fig. 65: Screenshot of the "View" menu.

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 stored with the image (separately from its metadata), 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 with the image and will be displayed in the main window.

> **Note**
>
> Annotations may be copied from an image to another by using the "Copy annotations" and "Paste annotations" actions of the "Edit Annotations" submenu (see Manipulate metadata and annotations).

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.

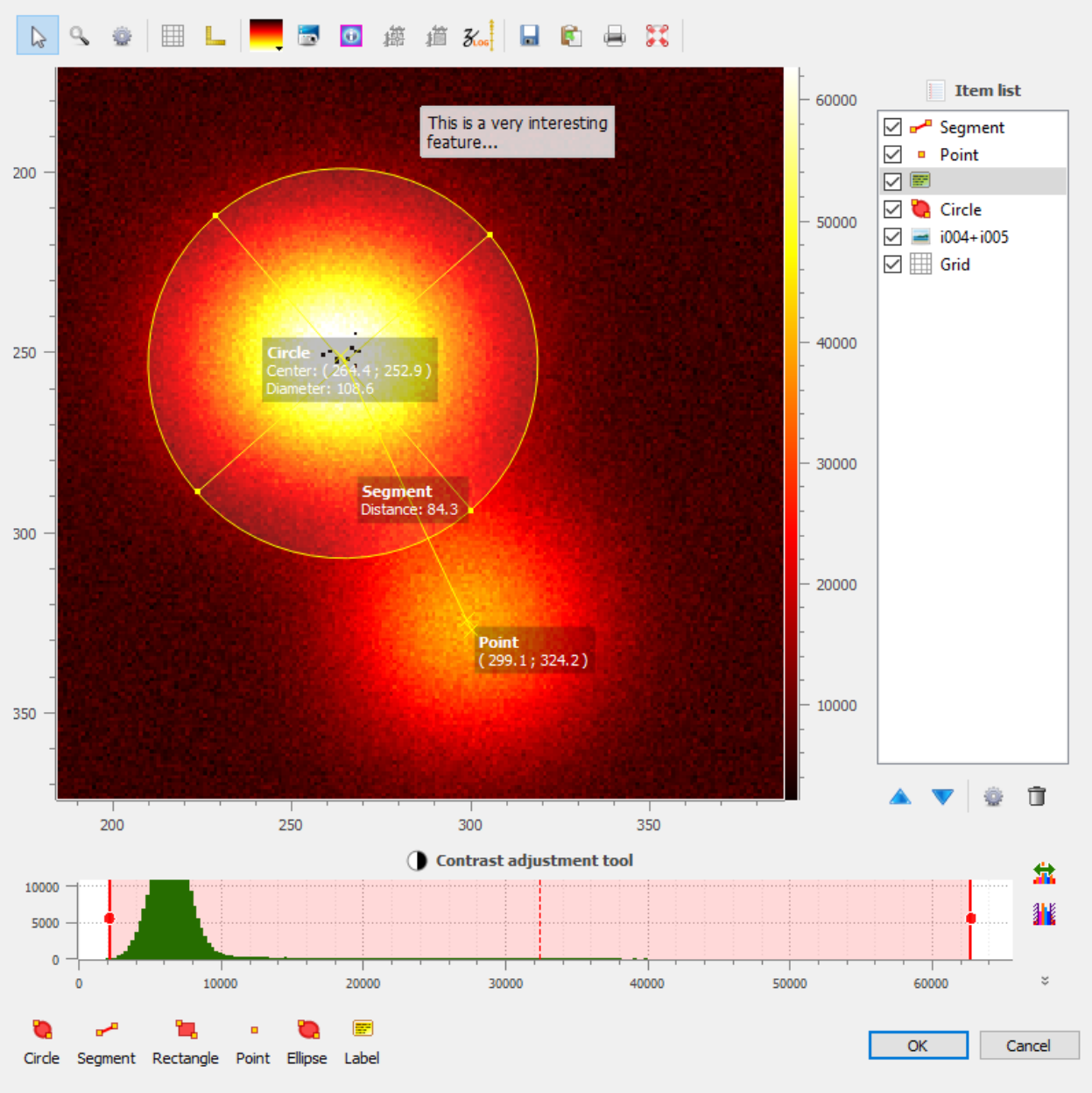


Fig. 66: Annotations may be added in the separate view.

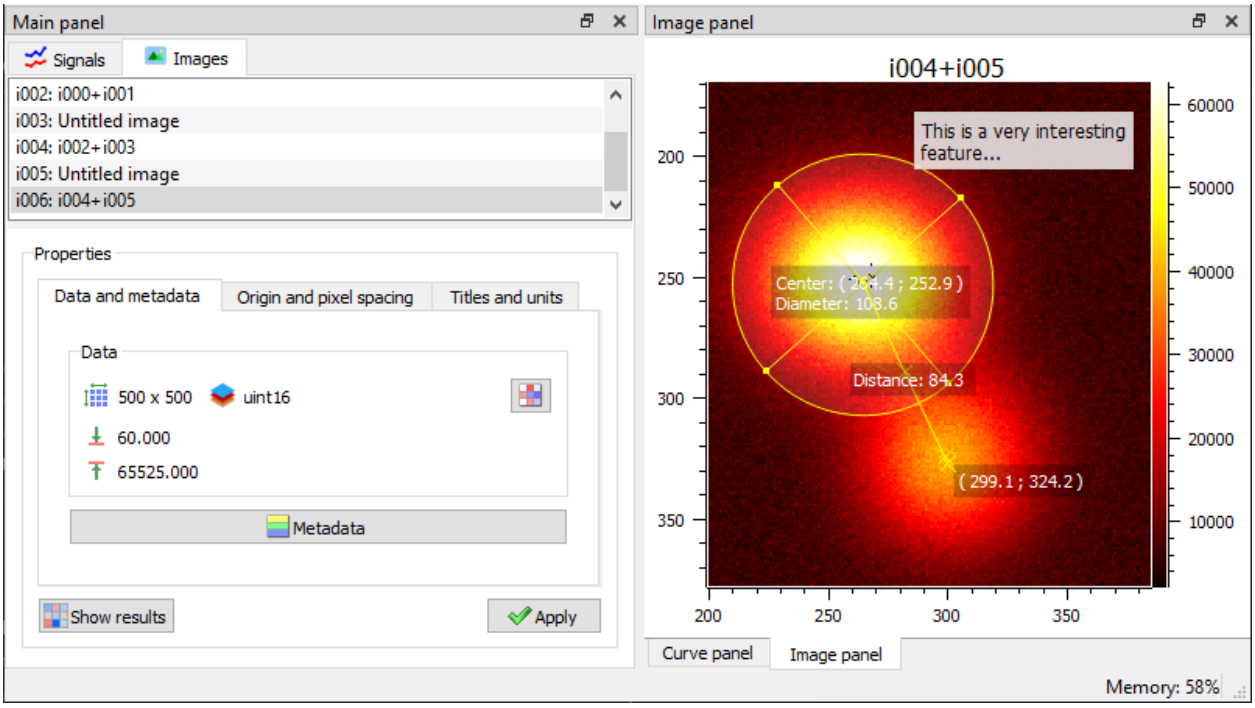


Fig. 67: Annotations are now displayed in the main window.

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 first object only

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.

Panels and toolbars

The last entries of the "View" menu show or hide the panels of the main window: the "Welcome" page (see Welcome page), the "Macro Panel" (see Macros), the "History Panel" (see History Panel), the "AI Assistant" (see AI Assistant), the "Applications" panel (see Applications panel), and the "Signal View" and "Image View" panels. They also show or hide the toolbars of the main window ("Main Toolbar", "Signal Panel Toolbar" and "Image Panel Toolbar").

3.3.9 2D Peak Detection

DataLab provides a "2D Peak Detection" feature which is based on a minimum-maximum filter algorithm.

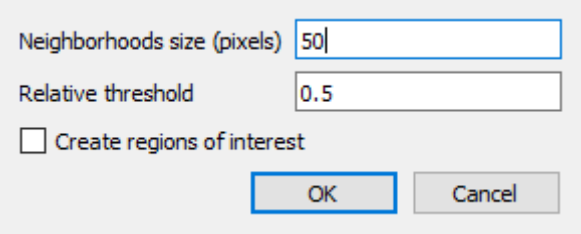


Fig. 68: 2D peak detection parameters.

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 threshold"
- 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, and is labelled with the name of the function and the reference of the image (followed by the ROI title, if a ROI is defined on input image)
- Columns x and y 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)
- 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:



Fig. 69: 2d peak detection results (see test “peak2d_app_test.py”)

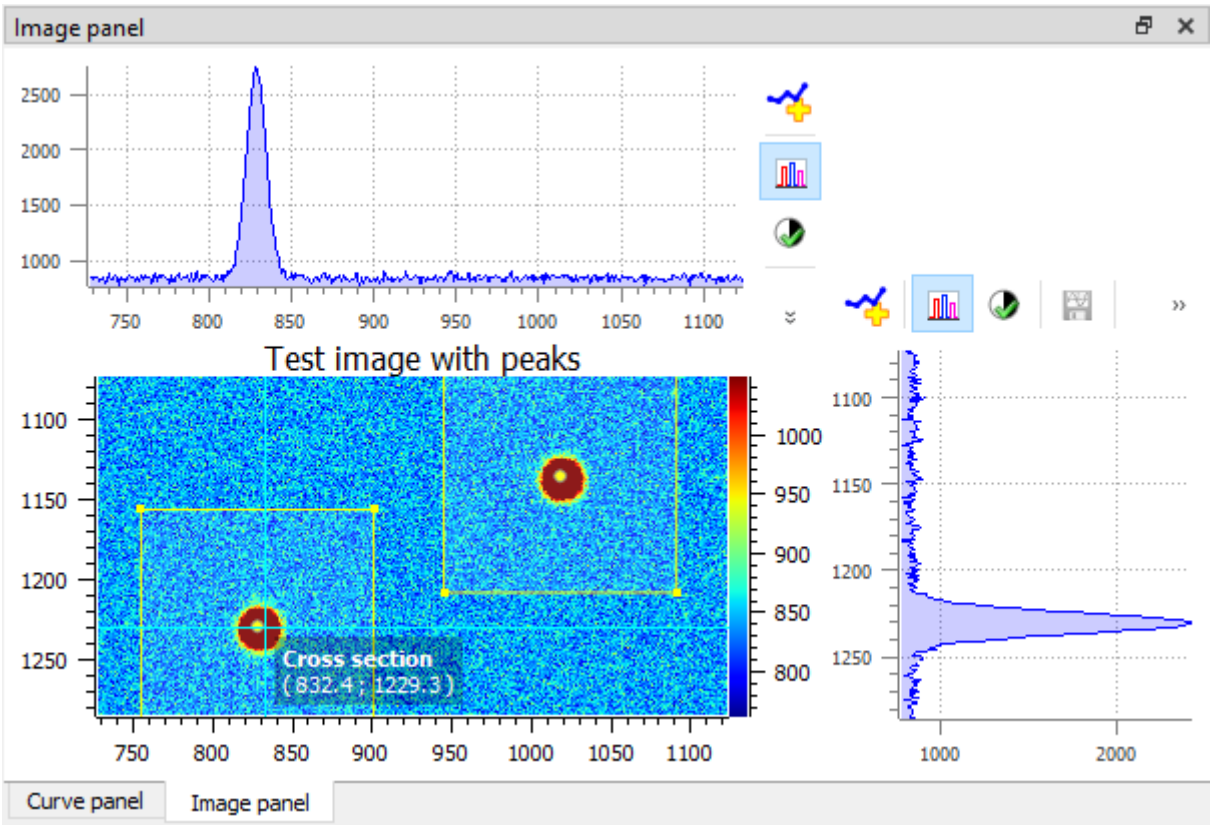


Fig. 70: Example of 2D peak detection.

- "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:

```
@warn_if_masked
@check_2d_array(non_constant=True)
def get_2d_peaks_coords(
    data: np.ndarray | ma.MaskedArray, 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)
    # Floating-point difference: the range of a signed integer image may␣
↪overflow
    data_diff = np.asarray(data_max, dtype=float) - data_min
    diff = data_diff > 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:
        # Eventually removing duplicates
        dist = distance_matrix(coords)
        for index in reversed(np.unique(np.where((dist < size) & (dist >␣
↪0))[1])):
```

(continues on next page)

(continued from previous page)

```
            coords.pop(index)
    return np.array(coords)
```

3.3.10 Contour Detection

DataLab provides a "Contour Detection" feature which is based on the marching squares algorithm.

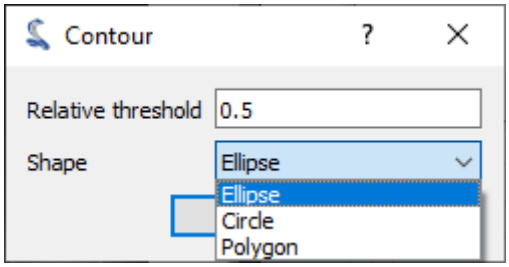


Fig. 71: Contour detection parameters.

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 parameters "Relative threshold" and "Shape" ("Ellipse", "Circle" or "Polygon")
- Optionally, enable "Create regions of interest" to automatically create a ROI from each detected contour:
  - ROI geometry is determined by the selected shape: circular ROIs for circles, polygonal ROIs for polygons and ellipses (approximated by a polygon)
  - Created ROIs can be useful for subsequent processing on each contour area



Fig. 72: Contour detection results (see test "contour_app_test.py")

Results are shown in a table:

- Each row is associated to a contour, and is labelled with the name of the function and the reference of the image (followed by the ROI title, if a ROI is defined on input image)
- Columns show contour coordinates: 3 columns for circles (center coordinates x, y and radius r), 5 columns for ellipses (center coordinates x, y, semi-axes a, b and angle θ), and vertex coordinates (x0, y0, x1, y1, …) for polygons

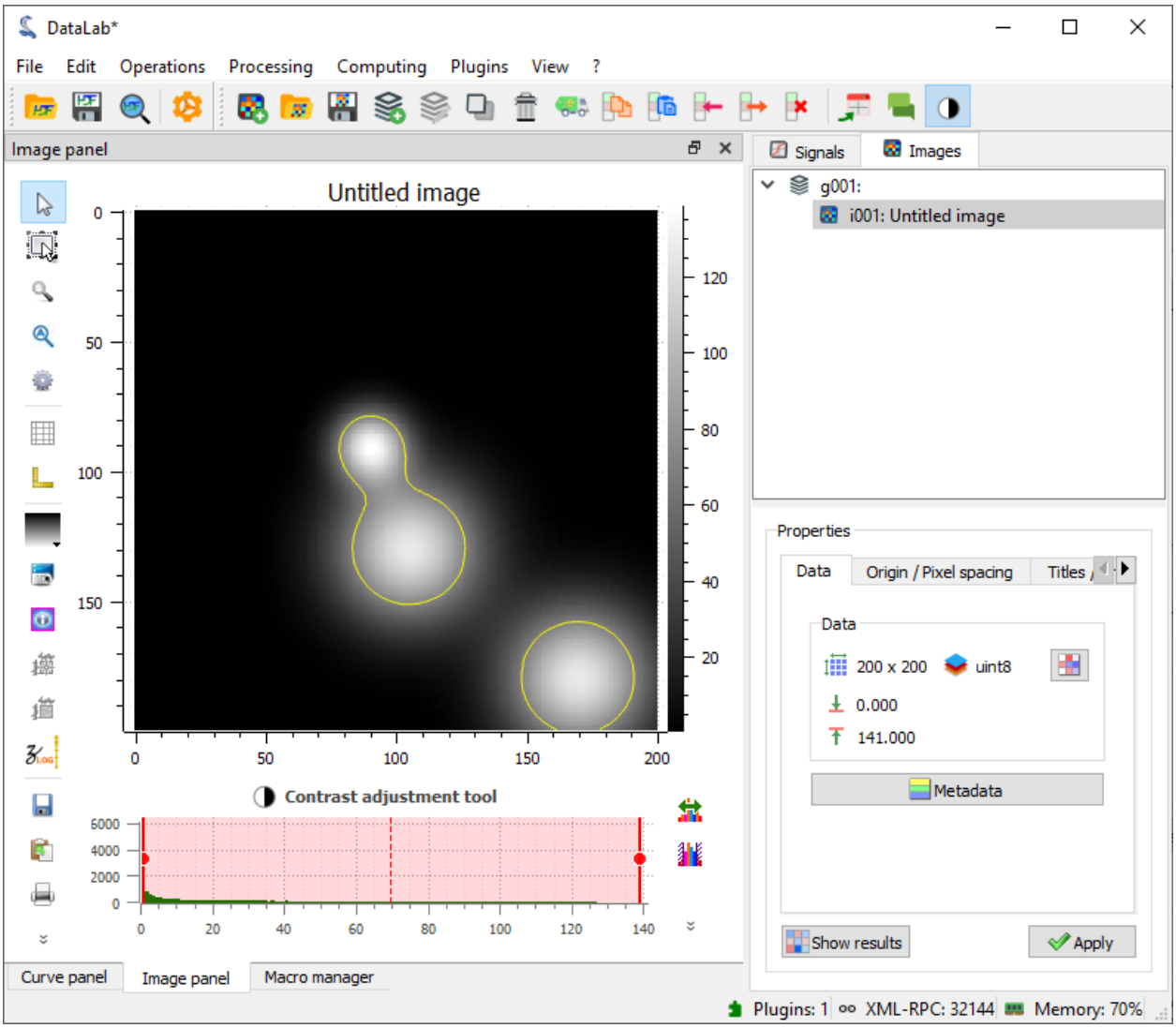


Fig. 73: Example of contour detection.

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:

```
def get_contour_shapes(
    data: np.ndarray | ma.MaskedArray,
```

(continues on next page)

(continued from previous page)

```
    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:
            result = fit_circle_model(contour)
            if result:
                xc, yc, r = result
                if r > 1.0:
                    coords.append([xc, yc, r])
        elif shape == ContourShape.ELLIPSE:
            result = fit_ellipse_model(contour)
            if result:
                xc, yc, a, b, theta = result
                if a > 1.0 and b > 1.0:
                    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 shape {shape}")
    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)
```

(continues on next page)

(continued from previous page)

```
        for i_row, coord in enumerate(coords):
            arr[i_row, : coord.shape[0]] = coord
        return arr
    return np.array(coords)
```

3.4 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).

3.4.1 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:

- 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

  The `sig`/`s` and `ima`/`i` short prefixes, and the `_unit` and `_validation` suffixes, are also accepted (e.g. `test_sig_<function_name>_unit`).

> **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 `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.

3.4.2 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.

3.4.3 Validation Status of DataLab

Functional validation

In DataLab, functional validation is based on a classic test strategy (see Functional validation).

Package	Coverage	Number of tests
DataLab	~82%	>1200
Sigima	~90%	>1600

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 (see the coverage of each package in the table above).

Table 4: Validation Statistics

Category	Signal	Image	Total
Number of compute functions	98	117	215
Number of validated compute functions	98	117	215
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 5: 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

continues on next page

Table 5 – continued from previous page

Compute function	Description	Test function
complex_from_magnitude_phase	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
extract_peak_positions	Extract peak positions as an XY-markers table	test_signal_extract_peak_positions
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

continues on next page

Table 5 – continued from previous page

Compute function	Description	Test function
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
replace_special_values	Replace NaN, +Inf and -Inf values in a signal	test_signal_replace_special_values
replace_x_by_other_y	Create a new signal using Y from src1 and Y from src2 as X coordinates	test_replace_x_by_other_y
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
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 6: 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_brightness_contrast	Apply a clipped linear brightness and contrast adjustment	test_adjust_brightness_contrast
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
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_phase	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

continues on next page

Table 6 – continued from previous page

Compute function	Description	Test function
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 the minimum physical circle enclosing pixel centers above midrange	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
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

continues on next page

Table 6 – continued from previous page

Compute function	Description	Test function
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, the image center or a given point	test_radial_profile
real	Compute real part	test_image_real
replace_special_values	Replace NaN, +Inf and -Inf values in an image	test_image_replace_special_values
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
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

continues on next page

Table 6 – continued from previous page

Compute function	Description	Test function
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

3.5 Advanced features

This section describes the advanced features of DataLab.

3.5.1 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.

A plugin is Python code loaded by DataLab: a single `datalab_<name>.py` file, or a package distributed as a wheel. Application plugins, for example for the characterization of cameras or of pulsed signals, or for the analysis of X-ray radiographs, add complete workflows to the Applications panel and to the welcome page.

Find plugins in the catalog

The DataLab plugin catalog lists the plugins maintained by the DataLab team (Official) and by their authors (Community). Each release is checked automatically before it is published: pure Python plugin code, open-source license, and dependencies provided by DataLab or, for DataLab Desktop, supplied as additional wheels or installed by users in DataLab's Python environment (see Additional dependencies and offline installation).

To install a plugin of the catalog, open Plugins > Configure plugins… in DataLab and select the Catalog tab (see Catalog tab). A plugin downloaded from the catalog page may also be installed from the From file tab, including in the standalone version of DataLab. The catalog indicates which plugins also run in DataLab-Web (see Plugins and DataLab-Web).

Your plugin could be the next one in the catalog: writing a plugin may take a single Python file, and publishing it takes a GitHub repository and one pull request, without any PyPI account. See the tutorial From an idea to the plugin catalog.

Plugin documentation

The plugin documentation is organized as follows:

- Installing and managing plugins: plugin locations, installation from the catalog or from a file, plugin configuration;
- Deploying plugins in an organization: local plugin catalogs, additional dependencies and offline deployment;
- Developing plugins: quick start, plugin generator, plugin API, examples and publication in the catalog;
- Application plugins: headless recipes, packaged examples, tools, instruments and welcome page tiles;
- Public API: reference of the classes and functions available to plugins.

Installing and managing plugins

This page describes where DataLab finds plugins, and how to install, enable, update and uninstall them. Plugins are offered by the DataLab plugin catalog; to distribute plugins within an organization, see Deploying plugins in an organization.

Where to put a plugin?

Plugins are automatically discovered at startup from multiple locations:

- An installed Python package may declare its plugin class through the standard `datalab.plugins` entry-point group:

  ```
  [project.entry-points."datalab.plugins"]
  my-plugin = "my_plugin.plugin:MyPlugin"
  ```

  The target must be a *datalab.plugins.PluginBase* subclass. This is the recommended distribution mechanism for plugins installed with `pip`.

- The user plugin directory: Typically ~/.DataLab_v1/plugins on Linux/macOS or C:/Users/YourName/.DataLab_v1/plugins on Windows.
- Additional plugin directories: Any number of extra directories may be declared from Plugins > Configure plugins…, in the Advanced tab (see Advanced tab). They are saved in the DataLab configuration and scanned at every startup.
- The standalone distribution directory: If using a frozen (standalone) build, the plugins folder located next to the executable is scanned.
- Installed plugins: Wheels and modules installed from Plugins > Configure plugins…, either from a file in the From file tab (see From file tab) or from a plugin catalog in the Catalog tab (see Catalog tab).
- The internal datalab/plugins/builtin folder (not recommended for user plugins): This location is reserved for built-in or bundled plugins and should not be modified manually.
- Additional directories listed in the `DATALAB_PLUGINS` environment variable: One or more directories may be specified, separated by the OS path separator (`;` on Windows, `:` on Linux/macOS), following the same convention as `PYTHONPATH`. All listed directories are appended to the plugin search path at startup; non-existent paths are silently skipped (a warning is written to the log file). Examples:

  ```
  # Linux/macOS
  export DATALAB_PLUGINS="/opt/my_plugins:/home/alice/datalab_plugins"
  ```

  ```
  :: Windows
  set DATALAB_PLUGINS=C:\my_plugins;D:\shared\datalab_plugins
  ```

  Changes to this variable are only taken into account at DataLab startup.

A plugin directory may also be the folder of a plugin project, such as one created by `datalab-plugin create --project`: DataLab then also searches its `src` subfolder, which holds the `datalab_*` folder of the plugin. The Python packages the plugin depends on must be available in DataLab's environment.

Managing plugins in DataLab

The Plugins menu provides two dedicated actions:

- Configure plugins… Opens the plugin configuration dialog, organized in four tabs: Installed plugins, Catalog, From file and Advanced. Hovering over a tab title describes its content. The dialog reopens with the size it had when last closed.
- Reload plugins Reloads plugin modules from disk without restarting DataLab.

When plugins are globally enabled, a single left click on the Plugins: label in the status bar also opens the same configuration dialog. The icon is not clickable; the shortcut remains available when no plugins are loaded or some plugins failed to load.

Installed plugins tab

This tab lists all discovered plugins with their icon, name, version, description and file path, as well as the plugins that failed to import. For each entry:

- an individual checkbox enables or disables the plugin,
- Open file opens the plugin source in your editor,
- Show in folder reveals the plugin file in the system file manager.

An Enable all plugins checkbox and a Filter combo box (All plugins, Enabled plugins, Disabled plugins, Plugins with errors) help navigating a large plugin collection. A Last loaded: … indicator shows when plugins were last loaded during the current session, and the Apply and reload plugins button applies the changes without leaving the dialog.

Catalog tab

This tab lists the plugins of the DataLab plugin catalog and of the local plugin catalogs (see Local plugin catalogs), with their description, license and source code link. Plugins maintained by the DataLab team are marked Official; the others are Community plugins, maintained by their authors, or Local catalog plugins, provided by your organization (the tooltip of such a plugin shows the folder of its catalog). A search field filters the list.

Install copies or downloads the wheel or single-file module available for Desktop and the running Python version, checks its size and SHA-256 digest, then installs it as described in From file tab, after confirmation. Versioned releases may offer Update to…. For a module without a comparable version, changed content offers Replace… instead; an identical digest is already installed. Plugins installed with `pip` or supplied by unmanaged source directories are identified but are not modified.

The catalogs are read only when the tab is shown or refreshed, and again when the local catalog folders change. A catalog that cannot be read, for example a network share that is not connected, is reported below the plugin count without hiding the plugins of the other catalogs. The address of the DataLab plugin catalog is the `plugins_catalog_url` option of the DataLab configuration file: it may point to a private copy of the catalog, or be emptied to disable it, for example on computers without Internet access. For a catalog stored in a folder, including a folder shared on the network, use a local plugin catalog instead.

From file tab

This tab installs a plugin shared as a file, without any Python tool: this is the way to add plugins to the standalone version of DataLab. Install from file… accepts two kinds of files:

- a wheel (`.whl`) built from a plugin project, for example with `python -m build`: it must be pure Python (`py3-none-any`), declare its plugin class in the `datalab.plugins` entry-point group, and have its requirements provided by the host or the adjacent dependency wheelhouse (see Additional dependencies and offline installation);
- a single Python module named `datalab_<name>.py`.

Before installing, DataLab checks the file without running it. For a wheel, the confirmation shows the distribution, version, entry points and requirements from its metadata. For a single-file module, it shows its import name, file, size, SHA-256 and any statically readable or explicit catalog metadata. An unknown version stays unknown and is displayed as a revision; no class or dependency list is guessed. A plugin runs with the same rights as DataLab: install only plugins from authors you trust.

Installed plugins are kept in the `installed_plugins` folder of the DataLab configuration directory (e.g. `~/.DataLab_v1/installed_plugins`) and listed in the tab, where Uninstall removes them. Modules can be reloaded after installation or replacement; a newly installed plugin is enabled. Additional dependencies and new versions of already loaded wheels are only activated at the next start of DataLab.

When reloading plugins, DataLab performs the following steps:

1. Unregister currently active plugins and their owned computations,

2. Clear plugin actions from signal and image panels,
3. Re-discover and reload plugin modules,
4. Re-register enabled plugins,
5. Re-register owned computations,
6. Recreate plugin actions and refresh menus.

This workflow allows iterative plugin development while DataLab is running.

> **Note**
>
> Plugin enable/disable state is persisted in DataLab settings. Disabled plugins remain listed in the configuration dialog and can be re-enabled later. The global third-party plugins setting in Preferences is also applied immediately: disabling it removes plugin actions and disables the status indicator, which displays Plugins: -, while enabling it reloads plugins automatically. The Plugins > Configure plugins… menu action remains accessible when the status indicator is disabled.

Advanced tab

This tab lists every directory scanned at startup, in two groups:

- Default plugin directories (read-only): the user plugin directory, the bundled datalab/plugins/builtin folder, the standalone distribution folder when applicable, and the directories declared through the `DATALAB_PLUGINS` environment variable (identified by a (from DATALAB_PLUGINS) suffix).
- Additional plugin directories: your own directories, added with the Add button and managed with the Edit directory and Remove directory buttons. Unlike the environment variable, these directories are saved in the DataLab configuration and therefore persist across sessions.

It also lists the Local plugin catalogs: folders, possibly shared on the network, whose plugins are offered in the Catalog tab (see Local plugin catalogs). They are managed with the same buttons, and the folders declared through the `DATALAB_PLUGINS_CATALOG` environment variable come first, as read-only entries. Changing these folders does not require reloading plugins.

The tab also holds a Compatibility warnings option to hide warnings for incompatible DataLab v0.20 plugins.

Deploying plugins in an organization

An organization may distribute its own plugins, or the versions of public plugins it has approved, from a folder shared on the network or copied to computers without Internet access. This page describes local plugin catalogs, plugins requiring additional packages and offline deployment. Users install these plugins from the Catalog tab, like those of the DataLab plugin catalog (see Catalog tab).

Local plugin catalogs

A local plugin catalog is a folder, typically shared on the network (for example `\\fileserver\datalab\plugins`), from which an organization distributes its own plugins, or the versions of public plugins it has approved, without publishing them. Local catalogs complement the DataLab plugin catalog and take precedence over it: when several catalogs list the same plugin (same distribution name or plugin ID), only the first one is shown, so that users install the version chosen by the organization. If the DataLab plugin catalog withdraws a plugin hidden this way, the tab reports it.

Local catalogs are declared:

- in the Advanced tab (see Advanced tab): these folders are saved in the DataLab configuration (`plugins_catalog_path_list` option);

- through the `DATALAB_PLUGINS_CATALOG` environment variable, for example to configure all the computers of an organization: folders are separated by the OS path separator (`;` on Windows, `:` on Linux/macOS) and take precedence over the folders of the configuration. Unlike `DATALAB_PLUGINS`, folders unavailable when DataLab starts are not ignored: they are reported in the Catalog tab, and used as soon as they become available and the catalogs are refreshed.

```
:: Windows
set DATALAB_PLUGINS_CATALOG=\\fileserver\datalab\plugins;D:\approved_plugins
```

A local catalog folder holds either:

- plugin files without an index: DataLab scans pure-Python wheels declaring a `datalab.plugins` entry point and source files named `datalab_<name>.py` at the folder root, without scanning subfolders. Wheels are grouped by distribution and version; each module is one installable file, even if it declares several plugin classes. Module metadata is read statically where possible; unknown names use the module name, and unknown versions remain empty and are tracked by digest. Invalid artifacts are reported. Copying a changed module or a new wheel version is enough to offer it;
- or a `catalog.json` file, in the format of the DataLab plugin catalog, whose release URLs are relative to the folder: only the plugins it lists are shown. It may be a copy of the DataLab plugin catalog with its wheels, or a catalog built from the wheels of the folder with `datalab-plugin catalog build`, then edited to give a display name to the plugins, deprecate or revoke them (`status` and `status_reason` fields), or withdraw a release (`yanked` field).

Whatever the catalog, DataLab checks the size and SHA-256 digest of the artifact before installing it, inspects it without running it and asks for confirmation; the dialog shows the catalog folder as the source. A module is checked for a valid importable file name, Python encoding and syntax, with a 4 MiB size limit. Inspection does not import its code or certify its behavior. The organization managing a shared catalog is responsible for its content and should restrict write access.

The `datalab-plugin` command maintains the `catalog.json` file of a folder:

```
:: Write catalog.json, listing the plugin files of the folder
datalab-plugin catalog build \\fileserver\datalab\plugins

:: Check catalog.json against the plugin files of the folder
datalab-plugin catalog check \\fileserver\datalab\plugins
```

`build` checks wheels and source modules without executing them and does not write the index if an artifact is refused. Wheels retain their pure-Python, entry-point and dependency checks; modules follow the rules below. Curated metadata, withdrawal reasons and revoked entries are preserved. The JSON file is replaced atomically. `check` reports missing, changed, refused or unlisted plugin files and dependency artifacts. Run `build` again after changing the root files. Both commands exit with a non-zero status when they find a problem.

Single-file modules in a catalog

For a simple offline catalog, put the unchanged `datalab_<name>.py` file at its root and add that folder under Local plugin catalogs. No `pyproject.toml`, wheel or Internet access is needed. To generate an index that an administrator can curate, run `datalab-plugin catalog build <folder>`, then `check`. Mixed folders may contain both modules and wheels:

```
approved_plugins\
    datalab_spectroscopy.py
    datalab_other_plugin-1.0-py3-none-any.whl
    catalog.json
    dependencies\
        additional_wheel_dependency-1.0-py3-none-any.whl
```

The generated JSON uses `kind: "module"` and the real import name in `module`; it does not give the script a fictitious Python distribution. Its release contains `filename`, a relative `url`, `size`, `sha256`, `targets: ["desktop"]` and `version` (possibly an empty string). `kind` is omitted or `"wheel"` for existing wheel entries. Add an optional `requires_python` constraint to a module release when its compatibility is known.

An administrator may edit the display name, summary, license and other descriptive fields in the JSON, and supply a catalog identity and version when they are not statically available in the source. `build` preserves these fields. A known source identity or version must agree with the catalog. For dynamic metadata or several plugin classes, the catalog describes the complete module without executing it. Its catalog identity does not rewrite the runtime `PluginInfo` or require editing a historical file.

If a module has no usable version, leave `version` empty: DataLab compares SHA-256 fingerprints instead. The confirmation and installed-files list distinguish a known version from a revision. A changed unversioned file, or different contents with the same declared version, requires explicit Replace… consent; it is not labelled as a newer version. Existing modules installed manually through DataLab are recognized too. Files in development directories are not overwritten; resolve a name conflict or remove the unmanaged copy before using the managed installation.

Single-file modules use packages already available to DataLab. Their `import` statements are not interpreted as package requirements, no dependencies are downloaded, and a `dependencies` subfolder is not installed for them. `prepare` includes modules in the catalog but performs dependency downloads only for wheels. Imports that cannot be checked statically may still fail when the plugin is loaded, and are then shown in the Installed plugins tab.

For the public catalog, publish the unchanged file as a named GitHub release asset and provide its version, license, description and SHA-256 in the YAML entry. The source is not required to expose those values statically. An entry has this structure (replace the example repository, file and fingerprint with the actual release information):

```yaml
id: io.github.alice.spectroscopy
name: Spectroscopy tools
kind: module
summary: A single-file Desktop plugin
license: BSD-3-Clause
repository: https://github.com/alice/spectroscopy
capabilities: [processing]
releases:
  - version: "1.0.0"
    filename: datalab_spectroscopy.py
    sha256: <SHA-256 of the released file>
    requires_python: ">=3.10"
```

The public catalog verifies and mirrors the script under `modules/<sha256>/...`. To use that published catalog offline, copy its JSON and complete relative tree, including both `modules` and `wheels` when present. Do not rebuild an unchanged mirror whose artifacts are in these subdirectories: the local authoring commands scan the root. Source URLs, revision checks, withdrawals and local-catalog priority follow the same rules for both formats. Module entries target Desktop only, not DataLab-Web or PyPI.

Additional dependencies and offline installation

Desktop plugins may require packages not shipped with DataLab. Keep the plugin itself as a pure-Python wheel, and supply the additional packages, including all their transitive dependencies, as ordinary wheels in a `dependencies` subfolder of the local catalog. These dependency wheels may contain native libraries, provided they match the Python version, ABI and platform of the host. They do not need a DataLab plugin entry point.

On a computer with Internet access, the authoring command prepares the wheelhouse and writes the catalog:

```
datalab-plugin catalog prepare D:\approved_plugins
```

The command downloads wheels for the current host without installing them or changing its packages. It uses DataLab's declared runtime dependencies as the reference, so that unrelated plugins already installed on the author's computer are not mistaken for packages shipped with DataLab. Prepare wheels on each target environment when different Python versions or platforms need different artifacts, then copy the complete folder to the offline computer or network share. `build` and `check` themselves do not download missing packages.

For example, a Windows CPython 3.11 catalog may contain:

```
approved_plugins\
    catalog.json
    datalab_tango-0.1.0-py3-none-any.whl
    dependencies\
        pytango-10.3.1-cp311-cp311-win_amd64.whl
        docstring_parser-0.18.0-py3-none-any.whl
```

The Catalog tab copies the verified dependency artifacts of the chosen release. For Install from file…, place the `dependencies` folder next to the plugin wheel. DataLab resolves the installation using only these local files, with no package-index fallback and no compilation. Before confirmation it displays the additional packages, versions, sizes, source folder and SHA-256 digests.

Packages already provided by the Python environment or by DataLab are never replaced, upgraded or downgraded. Additional packages are shared by plugins in the same Python runtime. An incompatible installed version, missing transitive wheel or unsupported native wheel causes an explicit refusal; no partial installation is activated. Catalog metadata checks do not replace this resolution against the actual host.

A catalog release may instead list packages that users install themselves in DataLab's Python environment (`environment_requirements`), for example a large native library installed with `pip` or `conda`: DataLab never downloads them. The Catalog tab shows these requirements and offers to install the plugin only once the Python environment satisfies them. The standalone application has no such environment: these plugins are shown as not available there, and can only be installed from a file with the wheels of the missing packages in a `dependencies` folder. When a wheel requires packages that are missing and no `dependencies` folder is available, DataLab names the missing packages and installs nothing.

An installation adding dependencies is activated after restarting DataLab. A newly installed plugin is then enabled, including when an explicit enablement list is configured. Uninstalling the plugin keeps the shared dependencies. Different Python/ABI/platform runtimes use separate dependency directories; dependencies incompatible with a changed host are reported rather than repaired over the network.

There is no environment per plugin: plugins requiring contradictory versions cannot coexist. Wheels requiring external system libraries or drivers still need those prerequisites installed separately. Standalone dependencies are intended for library imports, not for supplying an external Python interpreter or promising that their command-line scripts can be run independently. The Web installer remains unchanged.

On Windows, native library paths must also fit the DLL loader's path limit. DataLab uses short generation directory names and refuses an installation whose DLL paths would still be too long, before publishing it; use a shorter configuration directory in that case.

Offline deployment, step by step

This procedure is self-contained: the disconnected computer does not need a package index, a Python development environment, or access to the public plugin website. A local network share, USB drive or ordinary local folder works. Preparing missing wheel dependencies requires a connected reference computer; a catalog containing only source modules can be assembled entirely offline (see Single-file modules in a catalog).

Important

A plugin catalog is a source of installation files, not a plugin search directory. Add it under Local plugin catalogs, not Additional plugin directories. `DATALAB_PLUGINS_CATALOG` supplies catalogs; `DATALAB_PLUGINS` supplies code loaded directly. Installing copies the approved files into the user's DataLab configuration directory, so the source folder or network share is not needed to run the installed plugin afterwards.

1. Prepare a complete catalog on a connected computer

The provider needs DataLab installed in a Python environment, with the `datalab-plugin` command available. The standalone application does not supply this authoring command; its users only need the prepared files. Use a reference environment matching the target's DataLab version, Python version, operating system, architecture and host package versions. ? > Installation and configuration… identifies the target installation. A Windows 64-bit CPython 3.11 wheel is not interchangeable with a Linux, 32-bit or CPython 3.12 wheel.

Create a folder, place the plugin's `.whl` or `datalab_*.py` file at its root, then run:

```
datalab-plugin catalog prepare D:\approved_plugins
datalab-plugin catalog check D:\approved_plugins
```

For wheels, `prepare` resolves missing requirements against the reference host, downloads their wheels and transitive dependencies into `dependencies`, then writes `catalog.json`. It does not install them in the reference host or compile source distributions. Modules are included without dependency downloads. Use `build` for a purely file-based catalog; if preparation reports a missing wheel or a conflict, fix the catalog before transferring it.

Keep all files produced, including the dependency wheels. Do not copy just `catalog.json` or just the plugin wheel. If serving several target environments, prepare their wheels on matching reference environments, combine the needed files, then run `datalab-plugin catalog build D:\approved_plugins` followed by `check`. A pure-Python wheel may also restrict the supported Python versions in its metadata.

`check` checks the catalog files, their integrity and metadata. It is not a substitute for an installation test on the actual target: complete dependency resolution happens at installation time. Before wider deployment, test the complete folder on an equivalent installation with Internet access unavailable.

2. Transfer the folder, preserving its layout

Copy the complete folder to a USB drive, local disk or network share accessible from the target computer. Preserve relative paths and file names. A public mirror may store artifacts under `wheels\<digest>\...` and `modules\<digest>\...`: preserve those directories too, because `catalog.json` references them. The JSON file alone is not a mirror, and DataLab does not automatically export an online catalog for offline use.

For a strictly offline JSON catalog, the installation `url` of every release and dependency artifact must resolve to an existing local file. An absolute `https://` URL is still downloaded from its server even if the catalog itself is local; placing a wheel beside that JSON does not override its URL. Informational links such as `repository`, `documentation` and `source_url` do not replace installation files. Catalogs generated by `build` use relative file URLs; published mirrors must retain the matching relative directory tree.

The user needs read access to this source, and write access to their own configuration and temporary directories. Administrator privileges are not required to install a plugin this way. Limit write access to the shared catalog to its maintainers; a SHA-256 check detects changed files but does not certify that a plugin or dependency is safe.

3. Configure the disconnected computer

Open Plugins > Configure plugins… > Advanced. Under Local plugin catalogs, click Add, select the catalog root (the folder containing `catalog.json` or the plugin wheels), and apply the configuration. Do not select the `dependencies` subfolder. Open the Catalog tab and click Refresh if needed.

Administrators may instead set the following environment variable before launching DataLab:

```
$env:DATALAB_PLUGINS_CATALOG = 'D:\approved_plugins;\\fileserver\datalab\plugins'
```

This example applies to DataLab started from the same PowerShell window; a shortcut launched separately does not inherit that window's environment. An organization may also deploy the variable through its usual workstation configuration. Environment-variable folders come before folders saved in DataLab's settings; both come before the online catalog. The first matching plugin ID or distribution wins: releases from different sources are not merged.

> **Warning**
>
> Adding a local catalog does not disable the online catalog. If its URL is left configured, DataLab still tries to read it when refreshing, which can cause a network wait and an error on a disconnected computer. The other catalogs remain usable after that error.

To disable that lookup, close DataLab, back up its configuration file, and set `plugins_catalog_url` to an empty value in its existing `[main]` section. With the standard Windows user profile, DataLab 1.3 and later use `%USERPROFILE%\.DataLab_v1\DataLab_v1_typed.ini` (on Linux/macOS, normally `~/.DataLab_v1/DataLab_v1_typed.ini`). Custom profiles may use another base folder. Do not edit the legacy `DataLab_v1.ini` file instead.

```
[main]
plugins_catalog_url =
```

Preserve the rest of the file. Do not delete this option to disable it: an absent option uses the default public URL again. Restart DataLab after editing. A private HTTPS URL still requires access to that server; for a strictly file-based offline installation, use a local catalog folder.

4. Install, confirm and restart when requested

In the Catalog tab, find the plugin and click Install. DataLab chooses the newest release offered for Desktop and the running Python version, checks the plugin file and listed compatible dependency artifacts against their sizes and SHA-256 digests, then plans any additional installation using the local files only. Selection of a release does not guarantee that all its dependencies can be installed: this is checked against the actual host before confirmation.

Read the confirmation: it gives the plugin identity, classes, requirements and fingerprint, and lists the additional packages, their versions, sizes, fingerprints and source folder when needed. Cancel leaves the installation unchanged. If approved, DataLab installs into its private user storage; it does not modify the environment's existing package versions or load code from the shared catalog at every startup.

Restart DataLab if dependencies were added or if replacing an already loaded wheel. Otherwise, DataLab may offer a plugin reload. After restarting, check the Installed plugins tab for the plugin's state and any import error, then try one of its actions. The source catalog may be disconnected after successful installation. For a one-off file installation, From file > Install from file… uses the same checks; keep `dependencies` next to the plugin wheel.

5. Maintain and troubleshoot the offline catalog

When `catalog.json` exists, it is authoritative: DataLab lists and copies only its declared releases and dependency artifacts. For a catalog managed from files at its root, run `build` and `check` again after adding or changing a module, plugin wheel or dependency wheel, then redistribute the complete folder. Copying a new file beside an existing JSON does not update that index. `prepare` may require Internet for wheel dependencies; `build` and `check` never fetch missing packages.

Do not run `build` on an unchanged mirror whose plugins live only under `wheels\<digest>\...` or `modules\<digest>\...`: it scans the catalog root, not the recursive tree, and would remove unrepresented entries. Update a mirror by transferring a complete, consistent published catalog and its files. To manage it with the local authoring commands instead, first arrange plugin wheels and modules at the root and additional packages in `dependencies`.

Users refresh the Catalog tab to discover updates, then explicitly click Update to…. Updates are not installed automatically. Plugins installed with `pip` in the Python environment are identified but not updated by this dialog. Shared additional dependencies are retained when a plugin is uninstalled, and their existing versions remain protected on subsequent installations.

If an installation fails, read the error in the plugin dialog before changing files:

- Catalog not found or inaccessible: check the selected root, drive letter, USB drive, network connection and read permissions. A missing share is not an Internet problem. Refresh once it is available.
- Size or SHA-256 mismatch: the copied files differ from the catalog. Ask the provider for a complete, consistent copy or regenerate the catalog from the intended wheels; do not bypass the check.
- No matching wheel or missing requirement: the wheelhouse may lack a transitive dependency or the wheel for this Python/ABI/platform. Ask the provider to run `prepare` on a matching reference environment. The installer never falls back to PyPI.
- Incompatible installed version: the host or another plugin already provides a conflicting package version. Ask for a compatible release or reference installation; do not force an upgrade or downgrade to bypass the refusal.
- Windows DLL path too long: use a shorter configuration path. DLL import or missing system library: check the native package's external prerequisites with the provider; DataLab installs neither drivers nor system packages.
- Installed but not active: restart when requested, check the global and individual plugin enablement settings, and consult the import error displayed in the Installed plugins tab. Reloading alone does not replace an imported native library.

For local help, use ? > PDF documentation, then search for Offline deployment, step by step or `DATALAB_PLUGINS_CATALOG`. This PDF is a local file and does not need Internet access. Online documentation, the welcome page's documentation link, and external source-code/documentation links open websites instead. The PDF menu action appears only when a local PDF is present; DataLab tries the system language first, then English. If it is absent or unreadable, ask the administrator for the PDF shipped with the matching DataLab version rather than relying on an online link.

Developing plugins

This page explains how to write a plugin, from a single Python file to an installable project, and how to publish it in the DataLab plugin catalog. For a step-by-step walkthrough, see the tutorial From an idea to the plugin catalog.

Your first plugin in five minutes

The `datalab-plugin` command, installed with DataLab as a Python package, creates a working plugin in a single file:

1. Install DataLab in a Python environment, for example with `pip install datalab-platform` (see Installation). The standalone version of DataLab does not provide the `datalab-plugin` command.
2. Create the plugin, giving its name and a one-line description when asked:

```
datalab-plugin create
```

The command writes a `datalab_<name>.py` file to the user plugin directory.

3. Start DataLab, or select Plugins > Reload plugins if it is already running. The Plugins menu now holds a submenu named after the plugin, with a sample processing of the selected signals.
4. Open the file in your editor and change the `sample_processing` function, which receives the Y values of a signal as a NumPy array and returns new ones. Select Plugins > Reload plugins to try your changes, without restarting DataLab.

Then, you may:

- share the file with your colleagues, who install it with Plugins > Configure plugins… > From file (see From file tab);
- publish it in the DataLab plugin catalog (see Publishing a plugin in the catalog);
- for a larger plugin, with packaging and tests, create a layered project (see Creating a layered plugin project).

What is a plugin?

A plugin is Python code that is automatically loaded by DataLab at startup. It can define new features or modify existing ones.

A plugin defines a class that inherits from *`datalab.plugins.PluginBase`*, includes a class attribute named `PLUGIN_INFO` (an instance of *`datalab.plugins.PluginInfo`*) and implements the `create_actions` method. DataLab finds this class in a plugin directory (see Where to put a plugin?), in one of two forms:

- A single file whose name starts with `datalab_` (e.g. `datalab_myplugin.py`) and which contains the class. This is the simplest form, recommended to get started: `datalab-plugin create` creates such a file, ready to use (see Creating a plugin with datalab-plugin create).
- A folder whose name starts with `datalab_` and which holds an `__init__.py` file (a Python package, e.g. `datalab_myplugin/`). DataLab loads the folder, then its `adapters/desktop.py` file when it exists: this is where projects created by `datalab-plugin create --project` define the class (see Creating a layered plugin project). Otherwise, the class is defined in `__init__.py`.

Nothing needs to be installed: the file or folder is loaded as it is.

Plugins distributed as installed Python packages, such as the wheels of the plugin catalog, declare their `PluginBase` subclass through the `datalab.plugins` entry-point group, without relying on the `datalab_` prefix. A plugin found both ways, for example a project that is installed and whose folder is also a plugin directory, is loaded only once.

The `PLUGIN_INFO` object must define a unique, namespaced `id` that remains stable when the display name or implementation class changes. DataLab uses this ID for registration and persisted enablement settings. Plugins without an ID remain supported through a module-and-class fallback for backward compatibility, but new plugins should not rely on that fallback.

```
from datalab.plugins import PluginCapability, PluginInfo

PLUGIN_INFO = PluginInfo(
  id="org.example.my-plugin",
  name="My Plugin",
  version="1.0.0",
  capabilities=(
    PluginCapability.APPLICATION,
    PluginCapability.PROCESSING,
```

(continues on next page)

(continued from previous page)

```
    ),
)
```

`capabilities` declares how DataLab may present and consume a plugin. Supported values are `PROCESSING`, `IO`, `VISUALIZATION` and `APPLICATION`. A plugin may combine several values; domain applications typically declare `APPLICATION` together with `PROCESSING`. The plugin configuration dialog shows declared capabilities in a stable order. Existing plugins that omit the field remain valid and are shown without capability labels.

> **Note**
>
> Only files and folders whose names start with `datalab_` are 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.

Source organization and distribution

Source organization, distribution format and functional capabilities are independent concepts. A single-file plugin and a plugin organized as a Python package use the same *`datalab.plugins.PluginBase`* API; neither form determines whether it provides processing, input/output, visualization or application features.

- Single-file plugin: one `datalab_<name>.py` module, easy to edit and share without a packaging project. This historical form is also the default output of `datalab-plugin create`; it is not deprecated.
- Package or project: Python modules and, optionally, resources, tests and packaging metadata. `datalab-plugin create --project` offers one layered organization, not a mandatory architecture for all plugins.
- Wheel distribution: an installable `.whl` archive with standardized metadata and plugin entry points. A wheel can contain a single module or a larger package; it is a distribution format, not a functional plugin category.

Both single-file modules and wheels can be installed with Install from file… or distributed through online and local catalogs. The catalogs do not require the layered project generated by `--project`. Packaging a single module as a wheel remains possible, but is optional for Desktop.

Table 7: Development forms and distribution

Development form	Direct source loading	Managed distribution
Single Python file	`datalab_<name>.py` in a plugin directory	Raw `.py` file, or an optional wheel
Python package or project	Package folder, including a project's `src` layout	Wheel containing the package and its resources

Both managed formats work in local and online Desktop catalogs. The historical single-file form is not a separate functional plugin category, and its source does not need to be changed merely to list it in a catalog. DataLab-Web support is not implied by a Desktop catalog entry.

A directly loaded source file remains in its development directory. A file installed through DataLab is copied to its managed user storage. A plugin is not a macro: a plugin registers persistent application features, whereas a macro is a script run to automate a task. Supporting the historical single-file form does not mean that obsolete APIs from DataLab v0.20 are automatically supported.

Creating a plugin with `datalab-plugin create`

The `datalab-plugin` command is installed with DataLab as a Python package; the standalone version of DataLab does not provide it. Run it without options to create a single-file plugin:

```
datalab-plugin create
```

The command only asks for the plugin name and its description, then writes a `datalab_<name>.py` file to the user plugin directory (see Where to put a plugin?). Select Plugins > Reload plugins in DataLab, or restart DataLab: the plugin adds its submenu to the Plugins menu, with a sample processing of the selected signals. To implement your own processing, edit the `sample_processing` function of the file, which receives the Y values of a signal as a NumPy array and returns new ones, then reload the plugins.

Options adapt the generated file:

- `--object-kind image`: the sample processing applies to images, and its function receives the image data.
- `--capability`: without the `processing` capability, the submenu holds an action showing a message, to replace with your own.
- A destination folder, given as argument: it must then be added to the plugin directories (see Advanced tab).
- `--package` and `--plugin-id`: the module name and the plugin ID are otherwise derived from the plugin name. The module name must start with `datalab_`.

An existing file is never overwritten. To share the plugin, send its file: other users install it with Plugins > Configure plugins… > From file (see From file tab).

Both the raw single-file plugin and a wheel can be published in a Desktop catalog (see Single-file modules in a catalog). To develop a larger plugin with packaging and tests already configured, create a layered project with `--project` instead (see Creating a layered plugin project).

Creating a layered plugin project

With `--project`, the installed `datalab-plugin` command creates a small, installable project that follows the stable plugin and feature ownership contracts. Run it without other options for an interactive setup:

```
datalab-plugin create --project
```

The command asks for the display name and the GitHub account (user or organization) that will host the project, and offers defaults for the Python package, reverse-domain plugin ID, description, and destination. The default plugin ID is `io.github.<account>.<name>`; `org.datalab.` IDs are reserved for the DataLab-Platform organization, and `org.example.` is used when no account is given. For scripts or reproducible setup, pass the values explicitly:

```
datalab-plugin create --project datalab-camera-characterization \
  --name "Camera Characterization" \
  --github-account DataLab-Platform \
  --package datalab_camera_characterization \
  --plugin-id org.datalab.camera-characterization \
  --description "Characterize scientific cameras" \
  --capability application \
  --capability processing \
  --object-kind image
```

The generated `src`-layout project applies the architecture validated by the Camera pilot:

```
package root (host-independent identity)
├── core
├── workflow
└── adapters
    ├── desktop.py
    └── web.py
```

`core` holds host-independent domain behavior, `workflow` may use DataLab's headless recipe contracts, and `adapters` owns host integration. The `datalab.plugins` entry point targets the Desktop adapter; Web support starts explicitly as `unsupported` (see Plugins and DataLab-Web). A generated AST test prevents `core` and `workflow` from importing GUI, browser, or adapter modules.

The project also includes a stable *`datalab.plugins.PluginInfo`*, an owned sample processing when the `processing` capability is selected, Ruff settings, architecture and contribution documentation, a changelog, a README, and a BSD-3-Clause license. The command refuses to overwrite an existing destination.

The template remains deliberately small. Add domain recipes, DataSet parameters, packaged examples, and specialized CI only when the plugin needs them.

To try the plugin, add the generated directory as a plugin directory (see Advanced tab) and reload the plugins: DataLab loads the plugin class from `adapters/desktop.py` without installing the project. To validate the project, install it with its test dependencies and run its checks from the generated directory:

```
python -m pip install -e ".[test]"
python -m pytest
python -m ruff check .
```

The generated `pyproject.toml` requires Python 3.10 or later and the current DataLab major version (for example `datalab-platform >= 1.4, < 2`), and declares the project URLs when the GitHub account is known. The generated `.github/workflows/release.yml` publishes the plugin without any PyPI account: pushing a tag such as `v0.1.0` checks that the tag matches the version of `pyproject.toml` and of the package, runs the tests, builds the wheel, publishes it in a GitHub release with its SHA-256 checksum and its build provenance attestation, and prints the entry to submit to the DataLab plugin catalog, including the SHA-256 digest of the wheel. A tag such as `v0.2.0-rc1` (version `0.2.0rc1`) publishes a release candidate as a GitHub pre-release, without catalog entry. Users then install the wheel with Plugins > Configure plugins… > From file (see From file tab). Listing the plugin in the catalog is optional; once listed, it can also be installed from the Catalog tab (see Catalog tab).

Hot-reload workflow for plugin development

The hot-reload feature is designed to accelerate the plugin development cycle. Here is the recommended workflow:

1. Start DataLab normally.
2. Create or edit your plugin file (e.g. `datalab_myplugin.py`) in one of the plugin directories (e.g. `~/.DataLab_v1/plugins`).
3. In DataLab, use Plugins > Reload plugins to pick up your changes instantly.
4. Test your plugin actions directly in the running application.
5. Iterate: edit the file, reload, test — without restarting DataLab.

To selectively enable or disable specific plugins during development, use Plugins > Configure plugins… and its Apply and reload plugins button, so that changes take effect without leaving the dialog.

Plugin API helpers

Plugins inheriting from *datalab.plugins.PluginBase* have direct access to useful helpers:

- `self.signalpanel` and `self.imagepanel`: access to panel APIs and action handlers,
- `self.proxy`: a *datalab.control.proxy.LocalProxy* instance for object creation and processing,
- `show_warning`, `show_error`, `show_info`, `ask_yesno`: convenience dialog methods,
- `edit_new_signal_parameters` and `edit_new_image_parameters`: helpers for object parameter dialogs.

These helpers simplify plugin code and keep it consistent with DataLab behavior.

Processing plugins should override `register_computations()` and register each feature with a stable `feature_id` of the form `<plugin_id>:<local_id>` and `owner_plugin_id=self.plugin_id`. DataLab calls this hook after the signal and image panels exist. Owned features are removed automatically when the plugin is disabled, reloaded, or uninstalled. `create_actions()` may then reference the registered feature by its stable ID.

A registered feature runs a Sigima computation function, decorated with `sigima.proc.decorator.computation_function()`. For a 1-to-1 feature without parameters, the simplest is to wrap a function processing NumPy arrays with `sigima.proc.signal.Wrap1to1Func` (signal Y values) or `sigima.proc.image.Wrap1to1Func` (image data), as in the processing plugin example (see Example: processing plugin). DataLab refuses other functions when they are registered.

Live preview compatibility

> **Note**
>
> Standard parameterized functions registered with `register_1_to_1` can participate in live preview without a custom renderer. They must return a `SignalObj` or `ImageObj`, be importable in a spawned process, and accept serializable source/parameter copies. Preview uses a private process and does not publish results to the workspace. It is not a sandbox for filesystem, network or other external side effects. Register functions that must not run speculatively with `preview_enabled=False`. Serialization or computation failures are reported in the preview; the normal OK path remains available. Unknown custom `DataSet.edit` implementations and alternate guidata backends are not replaced.
>
> The dedicated preview process may be reused between dialogs. Plugin functions must not assume a fresh Python interpreter for each preview: process-local module state may persist until a preview is cancelled while running, plugins are reloaded, or DataLab exits.

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/builtin folder, the examples below or the plugins of the DataLab plugin catalog (see Plugins of the catalog).

> **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 parent directory containing the datalab package to your PYTHONPATH (e.g., using the PYTHONPATH Manager in Spyder). Install the project's dependencies separately.

> **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.

Plugins of the catalog

The plugins of the DataLab plugin catalog are open-source and show complete, tested implementations:

- Camera & Detector Characterization, Pulse & Transient Characterization and X-ray Radiography & NDT: application plugins with headless recipes, packaged examples, simulators and welcome page tiles (see Application plugins), for DataLab Desktop and DataLab-Web. Their `core`, `workflow` and `adapters` layers are those generated by `datalab-plugin create --project` (see Creating a layered plugin project).
- Synchrotron Beamline Data: a demonstration plugin adding file formats (SPEC and XDI files) and a beamline data analysis.
- TANGO Instruments: a demonstration plugin acquiring the attributes of TANGO devices in an instrument window, and requiring a package that users install in DataLab's Python environment.

Example: processing plugin

This example registers a processing feature with stable identity and ownership, then creates an action which dispatches it through the processor registry:

```
# -*- 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
```

(continues on next page)

(continued from previous page)

```
    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)
        mean_difference = np.mean(differences)
        if mean_difference == 0:
            return central_pixel
        weights = np.exp(-differences / mean_difference)
        return np.average(values, weights=weights)

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


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

    FEATURE_ID = "org.datalab.examples.custom-filters:weighted-average-denoise"
    PLUGIN_INFO = datalab.plugins.PluginInfo(
        id="org.datalab.examples.custom-filters",
        name="My custom filters",
        version="1.0.0",
        description="This is an example plugin",
        capabilities=(datalab.plugins.PluginCapability.PROCESSING,),
    )

    def register_computations(self) -> None:
        """Register computations owned by this plugin."""
        wrapped_func = sipi.Wrap1to1Func(weighted_average_denoise)
        self.imagepanel.processor.register_1_to_1(
            wrapped_func,
            "Weighted average denoise",
            feature_id=self.FEATURE_ID,
            owner_plugin_id=self.plugin_id,
        )
```

(continues on next page)

(continued from previous page)

```
    def create_actions(self) -> None:
        """Create actions"""
        acth = self.imagepanel.acthandler
        with acth.new_menu(self.PLUGIN_INFO.name):
            acth.action_for(self.FEATURE_ID)
```

Example: input/output plugin

Here is a simple example of a plugin that adds 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
    """

    HEADER = "<llllllffl"

    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
```

(continues on next page)

(continued from previous page)

```
        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,
                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)
```

(continues on next page)

(continued from previous page)

```
            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 ------------------------------------------------


class XYZImageFormat(SingleImageFormatBase):
    """Object representing Dürr NDT XYZ image file type"""

    FORMAT_INFO = FormatInfo(
        name="Dürr NDT",
        extensions="*.xyz",
        readable=True,
        writeable=True,
    )
```

(continues on next page)

(continued from previous page)

```
    @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())
```

Example templates used by the test suite

DataLab also provides plugin templates used by integration tests in `datalab/tests/features/plugins/templates`. They are useful as development references for:

- basic valid plugin structure,
- processing feature owned by a plugin (`register_computations`),
- nested plugin menus,
- plugins with dialog actions,
- plugins with many actions,
- plugins with long descriptions.

The corresponding feature tests are located in `datalab/tests/features/plugins/plugins_app_test.py` and cover plugin lifecycle, hot-reload behavior, error handling, duplicate names, and configuration filtering.

Other examples

Other examples of plugins can be found in the plugins/examples directory of the DataLab source code (explore here on GitHub).

Publishing a plugin in the catalog

Any author may list a plugin in the DataLab plugin catalog, so that DataLab users find it and install it from the Catalog tab (see Catalog tab). No PyPI account is needed:

1. Host the source code in a public GitHub repository, under an OSI-approved open-source license.
2. Attach the plugin file to a GitHub release: the unchanged `datalab_<name>.py` file of a single-file plugin, or the wheel of a plugin project, which the release workflow generated by `datalab-plugin create --project` builds and publishes.
3. Open a pull request adding one YAML file, which describes the plugin and its release, to the catalog repository. Its automatic checks download the plugin file, verify its SHA-256 digest and inspect it without running it.

Once the pull request is merged, the plugin appears on the catalog page and in the Catalog tab of DataLab, marked Community: the Official mark is reserved for the plugins of the DataLab team. The tutorial From an idea to the plugin catalog details each step, and the contribution guide gives the complete rules.

Plugins and DataLab-Web

DataLab-Web, the browser-native edition of the platform, provides its own implementation of the *`datalab.plugins.PluginBase`* API, which differs from the one of DataLab Desktop. A simple processing plugin using the shared API may run in both editions; plugins using host-specific features generally define a distinct class for each of them:

- One entry point per edition. A wheel declares its Desktop class in the `datalab.plugins` entry-point group and its DataLab-Web class in the `datalab.web_plugins` group: the browser only imports the classes of the `datalab.web_plugins` group. Both entry points may reference the same portable class or distinct classes, which may share the host-independent code of the plugin, such as the `core` and `workflow` layers of a project created by `datalab-plugin create --project`, whose `adapters/web.py` module starts as unsupported, without Web entry point:

```
[project.entry-points."datalab.plugins"]
my_plugin = "my_package.adapters.desktop:MyDesktopPlugin"

[project.entry-points."datalab.web_plugins"]
my_plugin = "my_package.adapters.web:MyWebPlugin"
```

- Processing registration and browser limitations. DataLab-Web supports `register_computations()` and `action_for()` for simple processing plugins, including those created by the single-file plugin generator. Desktop welcome page tiles (`WELCOME_TILES`) are ignored with a warning, and recipe launchers (`RECIPE_LAUNCHERS`) prevent the plugin from loading. Desktop methods for recipes or examples (`start_recipe`, `try_example`, `open_example`, etc.) are not available. A DataLab-Web class may declare headless recipes, examples and tools (`RECIPES`, `EXAMPLES` and `TOOLS`), which DataLab-Web presents and runs itself.
- Asynchronous dialogs are specific to DataLab-Web. In the browser, a dialog cannot block the event loop: DataLab-Web plugins use `async` action callbacks, `await param.edit_async(self.main)` and the `*_async` helpers of their base class (such as `show_info_async` or `edit_new_signal_parameters_async`). DataLab Desktop does not run `async` action callbacks (the action then does nothing), and its *`datalab.plugins.PluginBase`* class has no `*_async` helpers: Desktop plugins use synchronous callbacks and helpers.
- No Qt-only graphical interfaces. Plugins that embed custom Qt widgets or rely on PlotPy interactive tools are desktop-only; their graphical parts have no equivalent in the browser.

- Execution happens inside the browser (WebAssembly), with no native file-system access beyond the in-memory file system.

The DataLab plugin catalog indicates which of its plugins also support DataLab-Web; the Camera & Detector Characterization, Pulse & Transient Characterization and X-ray Radiography & NDT plugins are examples of plugins with a distinct DataLab-Web class. See the DataLab-Web plugin guide for the plugin API of the browser and for loading plugins in DataLab-Web.

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.

Application plugins

An application plugin, declaring the `APPLICATION` capability, provides complete workflows, such as the characterization of a camera or of pulsed signals, or the analysis of X-ray radiographs: its methods are presented in the Applications panel, and its tiles on the Welcome page. This page describes what a plugin may declare for this purpose: headless recipes and packaged examples, which any plugin may provide, then tools, instruments and welcome page tiles, which require the `APPLICATION` capability. The Camera & Detector Characterization, Pulse & Transient Characterization and X-ray Radiography & NDT plugins of the DataLab plugin catalog are complete examples.

This page describes DataLab Desktop. DataLab-Web runs the headless recipes, examples and tools declared by a distinct plugin class, but it does not support the Desktop methods launching recipes and examples, nor welcome page tiles (see Plugins and DataLab-Web).

Headless recipes

Plugins may expose versioned scientific workflows through the class-level `RECIPES` tuple. Each *datalab.plugins.recipes.RecipeDescriptor* declares a stable ID namespaced by the plugin ID, typed input slots, an optional guidata `DataSet` parameter class, and a headless callable:

Recipe descriptors remain owned by the current plugin class and are read through *datalab.plugins.PluginBase.get_recipes()*; they are not copied into a separate mutable registry. Reloading the plugin class therefore replaces its recipe declarations together with its implementation.

```
from datalab.plugins import PluginBase, PluginInfo
from datalab.plugins.recipes import (
  RecipeDescriptor,
  RecipeInputSlot,
  RecipeObjectOutput,
  RecipeOutcome,
)

def run_quick_check(inputs, parameters, context):
  source = inputs["source"][0]
  context.raise_if_cancelled()
  output = source.copy()
  context.report_progress(1.0, "Quick check complete")
  return RecipeOutcome(
    objects=(RecipeObjectOutput("checked-signal", output),),
  )

class MyPlugin(PluginBase):
  PLUGIN_INFO = PluginInfo(
```

(continues on next page)

(continued from previous page)

```
    id="org.example.my-plugin",
    name="My Plugin",
    version="1.0.0",
  )
  RECIPES = (
    RecipeDescriptor(
      recipe_id="org.example.my-plugin:quick-check",
      plugin_version="1.0.0",
      title="Quick check",
      version="1.0.0",
      inputs=(RecipeInputSlot("source", "signal", "one"),),
      run=run_quick_check,
    ),
  )

  def create_actions(self):
    pass
```

The descriptor's `plugin_version` must match `PLUGIN_INFO.version`. Keeping the plugin and recipe versions explicit lets execution records identify both the installed implementation and the independently versioned workflow.

DataLab validates the declarations of a plugin when loading it: recipes, recipe launchers, examples, tools and welcome tiles, described below. A plugin with an invalid declaration, for example a recipe whose `plugin_version` no longer matches the plugin version, is not loaded: its error is shown in the internal console and written to the log file, and the plugin is listed with this error in the Installed plugins tab of Configure plugins… (Plugins with errors filter). The other plugins are loaded normally.

The recipe callable receives a mapping from slot IDs to tuples of Sigima `SignalObj` or `ImageObj` instances, the parameter `DataSet` instance (or `None`), and a *`datalab.plugins.recipes.RecipeExecutionContext`*. The context provides technology-neutral progress and cancellation callbacks and has no Qt dependency.

When `parameter_class` is `None`, the recipe declares no parameters and the callable receives `None`. Otherwise, consumers must provide an instance of the declared `DataSet` subclass. The recipe runner enforces this rule before execution.

A *`datalab.plugins.recipes.RecipeOutcome`* contains named object outputs, structured diagnostics, and optional scalar results. A `TableResult` or `GeometryResult` is wrapped in *`datalab.plugins.recipes.RecipeResultOutput`* and uses `anchor_id` to reference the named object output that will own it. The contract validates these references without assigning DataLab workspace UUIDs. Workspace mutation and atomic commit are responsibilities of the recipe runner, not of the recipe callable.

Describing recipe inputs

DataLab shows users what each recipe expects and whether the current selection suits it. Describe every input slot accordingly:

```
from datalab.plugins.recipes import RecipeInputSlot, RecipeMetadataRequirement

FLAT_FRAMES = RecipeInputSlot(
  "flat_frames",
  "image",
  "many",
  title="Flat frames",
  description="Uniformly illuminated frames, two or more per exposure level",
  min_count=4,
```

(continues on next page)

(continued from previous page)

```
  metadata=(
    RecipeMetadataRequirement(
      "org.example.my-plugin.exposure_time_s", "Exposure time in seconds"
    ),
    RecipeMetadataRequirement(
      "org.example.my-plugin.frame_role", "dark or flat", required=False
    ),
  ),
)
```

`min_count` is the minimum number of objects bound to a `many` slot. Required metadata keys must be present on every bound object; optional keys are hints shown to the user. The recipe runner enforces both rules before the recipe callable is called.

Two optional hooks of *datalab.plugins.recipes.RecipeDescriptor* refine these declarations:

- `suggest_bindings(candidates)` returns a mapping from slot IDs to some of the candidate objects, for example to split frames by role from their metadata. Without it, compatible candidates go to the only slot accepting their type, and slots sharing a type are left for the user to assign.
- `check_inputs(inputs, parameters)` returns a list of *datalab.plugins.recipes.RecipeDiagnostic* values for fast, recipe-specific checks such as shapes, counts per level, or units. It must not compute results. Error diagnostics block the run and warnings are shown before it. An exception raised by the hook is reported as an `input-check-failed` error.

datalab.plugins.recipe_binding.assess_recipe_inputs() combines the slot declarations and both hooks. It returns a *datalab.plugins.recipe_binding.RecipeReadiness* with a status (`no_input`, `needs_assignment`, `not_ready`, `warnings` or `ready`), the proposed bindings, structured issues that hosts translate, and the recipe diagnostics. The module has no Qt dependency, so DataLab-Web uses the same code.

Running a recipe on Desktop

Most plugins do not call the runner directly. *datalab.plugins.PluginBase.start_recipe()* runs a recipe on the current selection with DataLab's generic launcher. It assesses the selection, opens an input assignment dialog only when the bindings are ambiguous or invalid, edits the declared parameters, and then calls the runner. A plugin menu action therefore only needs:

```
def run_quick_check(self):
  return self.start_recipe("org.example.my-plugin:quick-check")
```

The Run… buttons of the Applications panel use the same launcher. A plugin that needs a dedicated interface for one recipe may map its recipe ID to a plugin method name in the class-level `RECIPE_LAUNCHERS` mapping; the panel then calls this method, without arguments, instead of the generic launcher.

Use *datalab.gui.plugins.recipe_runner.RecipeRunner* from the GUI thread to run a recipe on explicit inputs. It validates inputs and parameters, executes the headless callable, and commits its outcome to the DataLab workspace:

```
from datalab.gui.plugins.recipe_runner import RecipeRunner

descriptor = self.get_recipes()[0]
outcome = RecipeRunner(self.main).run(
  descriptor,
  inputs={"source": (source_signal,)},
)
```

The runner rejects missing, extra, mistyped, or incorrectly sized input slots before recipe code is called. It also enforces `min_count`, required metadata and the error diagnostics of `check_inputs`, validates the optional `DataSet` instance and checks cancellation before execution, after execution, and immediately before commit. Recipe code must remain headless and must not mutate the workspace itself.

Only a validated *datalab.plugins.recipes.RecipeOutcome* reaches the commit phase. The Desktop runner creates one group per output panel, using the recipe title by default, then adds all signal and image outputs. Scalar result IDs are persisted as `<recipe-id>:<result-id>` function names on their named anchor objects, so several tables or geometries may coexist without metadata-key collisions. A failure during insertion removes every object and group created by that invocation and restores the previous workspace modified state and current panel.

Local execution provenance

After successful headless execution, the Desktop runner stores the same *datalab.plugins.recipes.RecipeRunRecord* in every output object's metadata. Its versioned, JSON-compatible payload contains a shared run UUID, plugin and recipe IDs and versions, resolved parameters as JSON, named input and output UUIDs, DataLab and Sigima versions, the completed status, and UTC start and finish timestamps.

The metadata survives JSON export and HDF5 workspace round-trips. The shared run UUID links outputs across the Signal and Image panels without adding group metadata or a global workflow history. Failed or cancelled invocations do not commit outputs and therefore leave no run record in the workspace.

Packaged examples

Plugins may expose native DataLab workspaces through the class-level `EXAMPLES` tuple. Each *datalab.plugins.examples.PluginExample* uses a `package:relative/path` resource instead of a development filesystem path:

```
from datalab.plugins.examples import PluginExample

class MyPlugin(PluginBase):
  EXAMPLES = (
    PluginExample(
      id="quickstart",
      title="Quick start",
      description="Small deterministic workspace",
      resource="datalab_my_plugin:examples/quickstart.h5",
      recipe_ids=("org.example.my-plugin:quick-check",),
      expected_checks=("summary-table",),
    ),
  )

  def create_actions(self):
    pass
```

The package must include the resource in its wheel. `resolve()` returns an `importlib.resources` traversable that also works for packages imported from a ZIP archive. For APIs requiring a filesystem path, use `as_file()` as a context manager and do not retain the returned path after the context exits.

datalab.plugins.PluginBase.get_examples() validates unique local IDs and recipe references. A registered plugin may call `open_example("quickstart")` to materialize and load a native DataLab HDF5 workspace. Opening clears the current workspace by default; pass `reset_all=False` to merge it instead.

`recipe_ids` lists the recipes an example is designed for. One example may serve several recipes, for instance a dark ramp analyzed by two methods. A plugin may also generate an example in memory by overriding `materialize_example()` to return a *datalab.plugins.examples.PluginExampleData*; its `parameter_values` mapping gives, for each recipe ID, the parameter values suited to the generated objects.

The Applications panel presents each recipe as a method with a live status for the current selection and a Run… button; its details give the expected inputs and the examples designed for it. Try with this example calls *datalab.plugins.PluginBase.try_example()*: it opens the example, prefills the recipe parameters with the example values, and runs the recipe once the user accepts them. Examples without `recipe_ids` are listed as datasets, which only open.

Plugin tools

A recipe covers a headless analysis run by DataLab. Any other interaction owned by an application plugin, such as a wizard, an interactive editor, a data preparation step or a simulated instrument, is a tool. Declare tools in the class-level `TOOLS` tuple of *datalab.plugins.tools.PluginTool* values:

```
from datalab.plugins.recipes import RecipeObjectType
from datalab.plugins.tools import PluginTool, ToolSelection

class MyPlugin(PluginBase):
  TOOLS = (
    PluginTool(
      id="annotate",
      title="Annotate frames...",
      launcher="annotate_frames",
      description="Set the acquisition metadata of the selected frames",
      object_type=RecipeObjectType.IMAGE,
      selection=ToolSelection.AT_LEAST_ONE,
    ),
  )

  def annotate_frames(self):
    ...
```

DataLab lists each tool in two places: in the Tools section of the Applications panel, and in the plugin's submenu of the Plugins menu, after the plugin's own actions. `object_type` gives the panel whose menu shows the tool; with `None`, both panels show it. `selection` states which selection the tool needs: `NONE` (the default), `EXACTLY_ONE`, `AT_LEAST_ONE` or `AT_LEAST_TWO` objects of `object_type`. When the selection does not match, DataLab disables the menu entry and the button of the Applications panel, and the button tooltip tells what to select. `icon` takes a `package:path` resource, a `module:filename` image registered with guidata (e.g. `datalab:delete.svg`) or a DataLab icon file name; by default, the tool uses the plugin icon. With `menu=False` (DataLab 1.4.1 or later), DataLab does not add the tool to the plugin's submenu: the plugin places it in one of its own submenus, and the tool is still listed in the Applications panel.

The launcher is called without arguments. *datalab.plugins.PluginBase.get_tools()* validates unique local IDs and the tool methods, and rejects tools declared by plugins without the `APPLICATION` capability. *datalab.plugins.PluginBase.assess_tool()* returns the reason why a tool cannot be opened for the current selection, or `None`.

Instruments

A tool may name an `instrument` method instead of a `launcher`. This method returns a *datalab.plugins.instruments.PluginInstrument*, and DataLab opens a window for it: a live view on the left, the instrument settings on the right, a Live button that refreshes the view continuously, a New group option and an Acquire button. The plugin writes no user interface code:

```
import guidata.dataset as gds
import numpy as np
from sigima.objects import create_signal

from datalab.plugins.instruments import (
```

(continues on next page)

(continued from previous page)

```
  InstrumentAcquisition,
  InstrumentFrame,
  PluginInstrument,
)

class SourceSettings(gds.DataSet):
  amplitude = gds.FloatItem("Amplitude", default=1.0, unit="V")
  count = gds.IntItem("Signals per acquisition", default=10, min=1)

class Source(PluginInstrument):
  live_interval_ms = 200

  def __init__(self):
    super().__init__(SourceSettings())

  def make_signal(self):
    x = np.linspace(0.0, 1.0, 500)
    y = self.settings.amplitude * np.sin(2 * np.pi * 5 * x)
    return create_signal("Source", x, y + np.random.normal(0.0, 0.05, x.size))

  def preview(self):
    signal = self.make_signal()
    return InstrumentFrame([signal], summary="...", value_range=(-2.0, 2.0))

  def acquire(self):
    signals = [self.make_signal() for _ in range(self.settings.count)]
    return InstrumentAcquisition("Source - acquisition 001", signals)

class MyPlugin(PluginBase):
  TOOLS = (
    PluginTool(
      id="source",
      title="Signal source...",
      instrument="signal_source",
      object_type=RecipeObjectType.SIGNAL,
    ),
  )

  def signal_source(self):
    return Source()
```

DataLab edits `settings` in place, with the active state, groups and tabs of the guidata DataSet, then calls *`preview()`* after each change and, in live mode, every `live_interval_ms` milliseconds. An *`InstrumentFrame`* holds signals drawn together or a single image, a short summary shown below the view, and an optional fixed Y range (signals) or color range (image). *`acquire()`* returns an *`InstrumentAcquisition`*: DataLab adds its objects to a new group of the matching panel, titled after the acquisition, and selects them, so that a recipe can run on them at once. Both methods raise `ValueError` with a user-facing message when the settings cannot be used; DataLab shows the message in the window.

When the user turns off the New group option of the window, DataLab adds the acquired objects to the current group of the panel instead: as they are selected, the next acquisitions go to the same group, until the user selects another group. The option and the current group are taken when Acquire is clicked: changing them while the acquisition is running does not change its destination, and a new group is created if the group was removed meanwhile. A new group is still created

when the panel has no current group, for example when it is empty. The *new_group* class attribute sets the default state of the option (`True` by default): an instrument whose acquisitions are usually single objects, such as a device attribute read on demand, may set it to `False`. The choice of the user, once made, is remembered for each instrument and prevails over this default.

DataLab creates the instrument once per tool and keeps it until the plugins are reloaded, so the settings remain from one opening to the next.

DataLab Desktop calls the instrument outside the user interface thread, one call at a time, even when its window is closed and opened again while a call is running, and changes its settings only between two calls: a slow instrument, such as a device waiting for its data, does not freeze DataLab. The optional *open()* and *close()* methods are called when the window opens and closes, including when plugins are reloaded or DataLab quits, for example to connect to a device and to release it. If `open` raises an error, its message stays in the window and the instrument is not called: the next refresh or acquisition, for example after a settings change, calls `open` again first. `close` is called even if `open` failed, and must release only what is still held. Data received by the instrument in its own threads, such as device events, is kept by the instrument for its next `preview` or `acquire` call.

Instrument connectors should attach an `experimental_source` metadata entry to acquired objects before returning them; DataLab does not fill it automatically. Use Sigima's `make_experimental_source` and `set_experimental_source` helpers to record the device address, attribute, selection and timestamps as appropriate. This identifies the measurement origin, not a reproducible acquisition. See the Sigima experimental data guide for an example and the portable metadata format.

DataLab-Web supports tools too: it lists them in the same places, opens instruments in the same window, and calls launchers in the plugin's Pyodide host.

Welcome page tiles

Every active plugin declaring the `APPLICATION` capability adds a tile to the Applications section, at the top of the Welcome page. By default, this tile is built from the plugin's `PluginInfo` name, description and icon. Clicking this tile shows the plugin page in the Applications panel.

The plugin icon is declared with `PluginInfo.icon`, either as a `package:relative/path` resource (SVG or bitmap image shipped in the plugin wheel), as a `module:filename` image registered by a module with guidata (e.g. `datalab:delete.svg`), or as the file name of a DataLab icon. It is shown on the default tile, in the Applications panel and in the plugin configuration dialog; plugins without icon use a generic plugin icon. To offer other entry points, for example to open an example directly, declare the class-level `WELCOME_TILES` tuple of *datalab.plugins.tiles.WelcomeTile* values:

```
from datalab.plugins import PluginBase, PluginCapability, PluginInfo
from datalab.plugins.tiles import WelcomeTile

class MyPlugin(PluginBase):
  PLUGIN_INFO = PluginInfo(
    id="org.example.my-plugin",
    name="My Plugin",
    icon="datalab_my_plugin:icons/my_plugin.svg",
    capabilities=(PluginCapability.APPLICATION,),
  )
  WELCOME_TILES = (
    WelcomeTile(id="application", title="My Plugin"),
    WelcomeTile(
      id="quickstart",
      title="Open quick start",
      description="Open the packaged quick start workspace",
      icon="datalab_my_plugin:icons/quickstart.svg",
```

(continues on next page)

(continued from previous page)

```
      launcher="open_quickstart",
    ),
  )

  def open_quickstart(self):
    self.launch_example("quickstart")

  def create_actions(self):
    pass
```

Declared tiles replace the default tile. The first one is the main tile of the plugin; the others are shown next to it when the Applications section has room for them, and otherwise become actions of its menu, opened from its "…" button or by right-clicking it. A tile without `launcher` shows the plugin page in the Applications panel; otherwise, the named plugin method is called without arguments. A tile without `icon` uses the plugin icon. *datalab.plugins.PluginBase.get_welcome_tiles()* validates unique local IDs and launcher methods, and rejects tiles declared by plugins without the `APPLICATION` capability. The welcome page refreshes its tiles when plugins are loaded, reloaded or disabled, and reports errors raised by a launcher without closing DataLab.

The user decides which applications are shown and in which order: the welcome page shows a limited number of tile rows, pinned applications first, then recently used ones, then the others by name (see Welcome page). A plugin therefore cannot rely on its tile being visible: its features must remain available from the Applications panel.

Welcome page tiles are specific to DataLab Desktop. DataLab-Web provides the `datalab.plugins.tiles` module for portable declarations, but ignores the tiles and reports a warning.

Public API

This page is the reference of the classes and functions available to plugins. See Developing plugins for their use, and Application plugins for the declarations of application plugins.

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.**ClassicsImageFormat**

Object representing classic image file types

static read_data(filename: str) → ndarray

Read data and return it

Parameters

filename – File name

Returns

Image array data

static write_data(filename: str, data: ndarray) → None

Write data to file

Parameters

- **filename** – File name
- **data** – Image array data

write_with_options(filename: str, obj: ImageObj, writer_options: dict[str, object]) → None

Write classic image data with validated format-specific options.

class datalab.plugins.**FailedPluginInfo**(name: str, filepath: str, traceback: str, source: str = '')

Information about a plugin that failed to load or instantiate.

class datalab.plugins.**FormatInfo**(name: str = None, extensions: str = None, readable: bool = False, writeable: bool = False, requires: list[str] = None)

Format info

to_rst_table_row() → str

Return reStructuredText table row for format info (table .. list-table:: format, with 5 columns)

class datalab.plugins.**ImageFormatBase**

Base image format object.

This class is used to define the interface for image I/O formats. It is an abstract base class that defines the methods that must be implemented by any image format class.

validate_read_param(param: DataSet | None) → None

Reject read parameters for formats that do not support them.

abstractmethod read(filename: str, worker: CallbackWorkerProtocol | None = None, * (Keyword-only parameters separator (PEP 3102)), param: DataSet | None = None) → Sequence[ImageObj]

Read list of image objects from file

Parameters

- **filename** – File name
- **worker** – Callback worker object
- **param** – Optional format-specific read parameters

Returns
List of image objects

abstractmethod write(filename: str, obj: ImageObj) → None

Write data to file

Parameters

- **filename** – file name
- **obj** – native object (signal or image)

Raises
NotImplementedError – if format is not supported

write_with_options(filename: str, obj: ImageObj, writer_options: dict[str, object]) → None

Write an image with format-specific options.

Parameters

- **filename** – File name
- **obj** – Image object
- **writer_options** – Format-specific writer options

Raises
ValueError – If this format does not support writer options

class datalab.plugins.**PluginBase**

Plugin base class

last_example_data: *PluginExampleData* | None = None

Data of the last opened generated example (None for packaged ones)

classmethod get_plugin_id() → str

Return the stable plugin ID or a deterministic legacy fallback.

property plugin_id: str

Return the stable plugin ID or a deterministic legacy fallback.

classmethod get_recipes() → tuple[RecipeDescriptor, ...]

Return validated recipe descriptors exposed by this plugin.

classmethod get_recipe(recipe_id: str) → RecipeDescriptor

Return one recipe descriptor by its namespaced ID.

classmethod get_recipe_launchers() → Mapping[str, str]

Return validated recipe-to-method bindings overriding the Desktop UI.

launch_recipe(recipe_id: str) → RecipeOutcome | None

Launch a recipe on the current selection.

A plugin method declared in RECIPE_LAUNCHERS replaces the generic interaction of *start_recipe()*.

get_selected_objects() → list[SignalObj | ImageObj]

Return the objects selected in the current signal or image panel.

example_parameter_values(recipe_id: str, objects: Sequence[SignalObj | ImageObj]) → Mapping[str, object]

Return the last example's parameter values for a recipe and objects.

Values are returned only when all objects come from the last opened generated example: they are suited to that data only.

assess_recipe(recipe_id: str, objects: Sequence[SignalObj | ImageObj] | None = None) → RecipeReadiness

Assess whether a recipe can run on objects (default: selection).

start_recipe(recipe_id: str, objects: Sequence[SignalObj | ImageObj] | None = None, parameter_values: Mapping[str, object] | None = None) → RecipeOutcome | None

Run a recipe through the generic DataLab interaction.

DataLab assigns the objects (default: selection) to the recipe inputs, asking the user when needed, checks them, edits the parameters, then runs the recipe.

Parameters

- **recipe_id** – namespaced recipe ID
- **objects** – candidate objects (default: current selection)

- **parameter_values** – initial parameter values (default: those of the last opened example, when the objects come from it)

Returns
Recipe outcome, or None if cancelled or failed

try_example(example_id: str, recipe_id: str) → RecipeOutcome | None

Open an example, then start one of the recipes it was designed for.

classmethod get_tools() → tuple[PluginTool, ...]

Return validated tools listed in the Applications panel.

classmethod get_tool(tool_id: str) → PluginTool

Return one tool by its plugin-local ID.

assess_tool(tool_id: str, objects: Sequence[SignalObj | ImageObj] | None = None) → str | None

Return why a tool cannot be opened on objects (default: selection).

Returns
Message telling which objects to select, or None if the tool can be opened

launch_tool(tool_id: str) → object

Open a plugin tool: call its launcher, or show its instrument.

Raises
ValueError – if the selection does not allow opening the tool

get_instrument(tool_id: str) → PluginInstrument

Return the instrument of a tool, created once per registration.

classmethod get_examples() → tuple[PluginExample, ...]

Return validated packaged examples exposed by this plugin.

classmethod get_example(example_id: str) → PluginExample

Return one packaged example by its plugin-local ID.

classmethod materialize_example(example_id: str) → PluginExampleData | None

Generate one example in memory, or defer to its package resource.

launch_example(example_id: str) → PluginExample | None

Confirm and open a generated or packaged example, then select it.

open_example(example_id: str, reset_all: bool = True) → PluginExample

Open a generated or packaged HDF5 example in the Desktop workspace.

Generated examples (`materialize_example()` returning data) are loaded directly into the signal/image panels; packaged examples are opened through their native HDF5 resource. The materialized data is kept in *last_example_data*, so that recipes started on these objects reuse the example's parameter values (see *example_parameter_values()*).

classmethod get_welcome_tiles() → tuple[WelcomeTile, ...]

Return validated welcome page tiles exposed by this application plugin.

Without declared `WELCOME_TILES`, an application plugin exposes one default tile built from its *PluginInfo*. Declared tiles without icon inherit the plugin icon.

launch_welcome_tile(tile_id: str) → object

Launch a welcome page tile.

A tile without launcher shows the plugin page in the Applications panel; otherwise, its plugin method is called without arguments.

Returns
Value returned by the launcher method (None for the Applications panel)

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

The declarations of the plugin (recipes, examples, tools, welcome tiles) are validated first: an invalid plugin is not registered, so that the Applications panel, the welcome page and the menus only get valid declarations.

Raises

- **TypeError** – if a declaration has the wrong type
- **ValueError** – if a declaration is invalid or the plugin ID is already used

unregister()

Unregister plugin

remove_owned_features() → None

Remove this plugin's computing features from available processors.

register_hooks()

Register plugin hooks.

Called by *register()* during application startup or plugin reload, before the panels' PLUGINS action category exists: use this for non-GUI side effects only (I/O format registration, listeners). `self.main` and `self.proxy` are already set. Raising here rolls back the whole registration (the plugin stays unregistered).

unregister_hooks()

Unregister plugin hooks.

Called by *unregister()*; must undo *register_hooks()*. Owned computing features are removed automatically afterwards.

register_computations() → None

Register owned computations after signal and image panels exist.

Called once per registered plugin, right before *create_actions()*; raising here removes the plugin's owned features but keeps the plugin registered.

abstractmethod create_actions()

Create actions.

Called immediately after *register_computations()*, inside the panels' PLUGINS action category: actions created here populate the Plugins menu.

class datalab.plugins.**PluginBaseMeta**(name, bases, namespace, / (Positional-only parameter separator (PEP 570)), **kwargs)

Mixed metaclass to avoid conflicts

class datalab.plugins.**PluginCapability**(value)

Capability exposed by a plugin to DataLab consumers.

class datalab.plugins.**PluginInfo**(name: str = None, version: str = '0.0.0', description: str = '', icon: str = None, id: str | None = None, capabilities: Collection[PluginCapability] = <factory>, documentation_url: str | None = None)

Plugin info

name

Plugin display name

Type
str

version

Plugin version

Type
str

description

Plugin description

Type
str

icon

Plugin icon: `package:relative/path` resource, `module:filename` image registered with guidata, or DataLab icon file name

Type
str

id

Stable, namespaced plugin ID (e.g. `"org.example.my-plugin"`)

Type
str | None

capabilities

Plugin capabilities (*`PluginCapability`* values)

Type
collections.abc.Collection[datalab.plugins.base.PluginCapability]

documentation_url

HTTP or HTTPS URL of the plugin documentation, opened by the Documentation action of the Applications panel

Type
str | None

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 a plugin by stable ID, legacy name, or class.

Legacy display names are accepted only when they identify a single plugin.

classmethod register_plugin(plugin: PluginBase)

Register plugin

classmethod unregister_plugin(plugin: PluginBase)

Unregister plugin

classmethod unregister_all_plugins()

Unregister all plugins

classmethod clear_plugin_classes() → None

Clear registered plugin classes.

This is mainly useful when reloading plugin modules at runtime.

classmethod add_discovery_error(tb_text: str) → None

Record an error traceback that occurred during plugin discovery.

Parameters
tb_text – Formatted traceback string

classmethod get_discovery_errors() → list[str]

Return error tracebacks collected during plugin discovery.

Returns
List of formatted traceback strings (may be empty)

classmethod clear_discovery_errors() → None

Clear recorded discovery errors.

classmethod add_failed_plugin(name: str, filepath: str, tb_text: str, source: str = '') → None

Record a plugin that failed to load or instantiate.

Parameters

- **name** – Module or plugin class name
- **filepath** – File path of the plugin module
- **tb_text** – Formatted traceback string
- **source** – Discovery source description

classmethod get_failed_plugins() → list[FailedPluginInfo]

Return structured info about plugins that failed to load.

Returns
List of FailedPluginInfo objects (may be empty)

classmethod clear_failed_plugins() → None

Clear recorded failed plugin info.

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.**SignalFormatBase**

Class representing a signal file type

static create_object(filename: str, index: int | None = None) → SignalObj

Create empty object

Parameters

- **filename** – File name
- **index** – Index of object in file

Returns
Signal object

create_signal(xydata: ndarray, filename: str, index: int | None = None) → SignalObj

Create signal object from xydata and filename.

Parameters

- **xydata** – XY data

- **filename** – File name
- **index** – Index of object in file

Returns
Signal object

create_signals_from(xydata: ndarray, filename: str) → list[SignalObj]

Create signal objects from xydata and filename

Parameters

- **xydata** – XY data
- **filename** – File name

Returns
List of signal objects

read(filename: str, worker: CallbackWorkerProtocol | None = None) → Sequence[SignalObj]

Read list of signal objects from file

Parameters

- **filename** – File name
- **worker** – Callback worker object

Returns
List of signal objects

read_xydata(filename: str) → ndarray

Read data and metadata from file, write metadata to object, return xydata

Parameters
filename – File name

Returns
XY data

datalab.plugins.**discover_plugins**() → list[ModuleType]

Discover plugins through package entry points and naming convention.

Installed packages may expose a *PluginBase* subclass through the `datalab.plugins` entry-point group. This function also reloads or imports modules and packages matching the historical `"{MOD_NAME}_*"` naming scheme, as well as the `adapters.desktop` module of such packages, which defines the plugin class of packages laid out by `datalab-plugin create`.

Import errors for individual plugins are captured and logged so that one broken plugin does not prevent the others from loading. Error tracebacks are accumulated in *PluginRegistry* class attributes so that callers (e.g. the main window) can replay them into the internal console once it is ready.

This function mutates `sys.modules` and the plugin registry without synchronization: call it from the Qt main thread only.

Returns
Imported/reloaded modules containing discovered plugins

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.**format_tool_requirement**(tool: PluginTool) → str

Return the message telling which objects to select to open a tool.

datalab.plugins.**get_available_plugins**() → list[PluginBase]

Instantiate and get available plugins

Returns
List of available plugins (as instances)

datalab.plugins.**migrate_enabled_plugin_ids**(enabled_plugins: list[str] | None, plugin_classes: list[type[PluginBase]] | None = None) → list[str] | None

Migrate enabled plugin display names to stable IDs.

Unknown entries are retained so temporarily unavailable plugins do not lose their activation state. A legacy name shared by multiple plugins enables all matching IDs because the old setting cannot distinguish between them.

Parameters

- **enabled_plugins** – Configured stable IDs or legacy display names
- **plugin_classes** – Classes available for migration (registry by default)

Returns
Enabled stable IDs and retained unknown entries, or None for all plugins

datalab.plugins.**reload_plugin_modules**() → None

Reload plugin modules and reset plugin classes.

This helper is intended for hot-reloading plugins at runtime. It:

- Updates the plugin search path
- Clears the plugin class registry
- Reloads or imports all modules matching the plugin naming convention

Like *discover_plugins()*, this is not thread-safe: call it from the Qt main thread only.

datalab.plugins.PLUGIN_ENTRY_POINT_GROUP is the entry-point group of DataLab Desktop plugin classes ("datalab.plugins"); DataLab-Web plugin classes use the datalab.web_plugins group (see Plugins and DataLab-Web). The other names exported by *datalab.plugins*, such as PluginRegistry or discover_plugins, are used by DataLab to load and manage plugins, and are not described here. Headless contracts for plugin-provided scientific recipes.

exception datalab.plugins.recipes.**RecipeCancellationError**

Raised when recipe execution observes a cancellation request.

class datalab.plugins.recipes.**RecipeCardinality**(value)

Number of objects accepted by a recipe input slot.

class datalab.plugins.recipes.**RecipeDescriptor**(recipe_id: str, plugin_version: str, title: str, version: str, run: Callable[[Mapping[str, tuple[SignalObj | ImageObj, ...]], DataSet | None, RecipeExecutionContext], RecipeOutcome], description: str = '', inputs: Sequence[RecipeInputSlot] = (), parameter_class: type[DataSet] | None = None, suggest_bindings: Callable[[tuple[SignalObj | ImageObj, ...]], Mapping[str, Sequence[SignalObj | ImageObj]]] | None = None, check_inputs: Callable[[Mapping[str, tuple[SignalObj | ImageObj, ...]], DataSet | None], Sequence[RecipeDiagnostic]] | None = None)

Versioned, typed declaration of a plugin-provided headless recipe.

A missing `parameter_class` declares a parameterless recipe whose run callable receives `None` as its second argument. Otherwise, that argument must be an instance of the declared `DataSet` subclass.

`suggest_bindings` receives the candidate objects and returns a proposed assignment of some of them to input slots (e.g. from metadata). Hosts use it to bind inputs without asking the user.

`check_inputs` receives complete inputs and parameters (defaults when assessing a selection) and returns diagnostics: errors block the run, warnings are shown. It must stay fast (metadata, shapes, counts) because hosts call it whenever the selection changes.

property plugin_id: str

Return the owning plugin ID encoded in the recipe ID.

property local_id: str

Return the recipe-local ID encoded in the recipe ID.

get_input(slot_id: str) → RecipeInputSlot

Return one declared input slot by its ID.

class datalab.plugins.recipes.**RecipeDiagnostic**(level: RecipeDiagnosticLevel, code: str, message: str, details: Mapping[str, object]=<factory>)

Structured, machine-identifiable diagnostic emitted by a recipe.

class datalab.plugins.recipes.**RecipeDiagnosticLevel**(value)

Severity of a structured recipe diagnostic.

class datalab.plugins.recipes.**RecipeExecutionContext**(progress_callback: Callable[[float, str | None], None] | None = None, cancellation_callback: Callable[[], bool] | None = None)

Technology-neutral progress and cancellation callbacks for a recipe.

property is_cancelled: bool

Return whether cancellation has been requested.

raise_if_cancelled() → None

Raise *RecipeCancellationError* when cancellation is requested.

report_progress(progress: float, message: str | None = None) → None

Report normalized progress between zero and one.

class datalab.plugins.recipes.**RecipeInputSlot**(id: str, object_type: RecipeObjectType, cardinality: RecipeCardinality, required: bool = True, title: str = '', description: str = '', min_count: int = 1, metadata: Sequence[RecipeMetadataRequirement] = ())

Typed input slot declared by a recipe.

Parameters

- **id** – slot ID, local to the recipe
- **object_type** – type of the accepted objects
- **cardinality** – one object, or several objects
- **required** – True if the slot cannot be left empty
- **title** – human-readable slot name (derived from `id` if empty)

- **description** – what the objects must represent and how they are acquired
- **min_count** – minimum number of objects of a non-empty `MANY` slot
- **metadata** – metadata entries expected on every bound object

property display_title: str

Return the declared title, or one derived from the slot ID.

class datalab.plugins.recipes.**RecipeMetadataRequirement**(key: str, description: str = '', required: bool = True)

Metadata entry expected on the objects bound to a recipe input slot.

Parameters

- **key** – metadata key
- **description** – meaning of the value (unit, allowed values…)
- **required** – True if objects lacking the key cannot be analyzed; False documents an optional hint, e.g. used to assign objects to slots

class datalab.plugins.recipes.**RecipeObjectOutput**(id: str, value: SignalObj | ImageObj)

Named signal or image produced by a recipe.

class datalab.plugins.recipes.**RecipeObjectType**(value)

Scientific object type accepted by a recipe input slot.

class datalab.plugins.recipes.**RecipeOutcome**(objects: Sequence[RecipeObjectOutput] = (), results: Sequence[RecipeResultOutput] = (), diagnostics: Sequence[RecipeDiagnostic] = ())

Objects, anchored scalar results, and diagnostics produced by a recipe.

class datalab.plugins.recipes.**RecipeResultOutput**(id: str, value: TableResult | GeometryResult, anchor_id: str)

Named scalar result attached to a named object output.

class datalab.plugins.recipes.**RecipeRunRecord**(run_id: str, plugin_id: str, plugin_version: str, recipe_id: str, recipe_version: str, parameters_json: str, input_uuids: Mapping[str, Sequence[str]], output_uuids: Mapping[str, str], datalab_version: str, sigima_version: str, status: RecipeRunStatus, started_at: str, finished_at: str, schema_version: int = 1)

Versioned local provenance shared by every output of a recipe run.

The record is persisted as JSON metadata (see *to_dict()*) under the `RECIPE_RUN_RECORD_OPTION` metadata option of every output object, so external consumers (DataLab-Web, HDF5 workspaces, tooling) can rely on the following schema (version `1`):

- `schema_version` (int): always `RECIPE_RUN_RECORD_SCHEMA_VERSION`
- `run_id` (str): UUID shared by all outputs of one recipe invocation
- `plugin_id` / `plugin_version` (str): owning plugin and its version
- `recipe_id` / `recipe_version` (str): recipe (`<plugin_id>:<name>`) and its contract version
- `parameters_json` (str): guidata DataSet serialized with `dataset_to_json` (`"null"` for parameterless recipes)
- `input_uuids` (dict[str, list[str]]): object UUIDs consumed per input slot — a mapping (not a flat list) so slot roles survive round-trips

- `output_uuids` (dict[str, str]): object UUID produced per output ID
- `datalab_version` / `sigima_version` (str): environment versions
- `status` (str): a `RecipeRunStatus` value
- `started_at` / `finished_at` (str): timezone-aware ISO 8601

Compatibility policy: readers must reject records whose `schema_version` they do not know (as *`from_dict()`* does); any field addition, removal, or semantic change requires bumping `RECIPE_RUN_RECORD_SCHEMA_VERSION` and providing an explicit migration path for persisted workspaces.

to_dict() → dict[str, object]

Return a representation composed only of JSON-compatible values.

classmethod from_dict(data: Mapping[str, Any]) → RecipeRunRecord

Restore a recipe run record from persisted metadata.

class datalab.plugins.recipes.**RecipeRunStatus**(value)

Terminal status persisted on successful recipe outputs.

exception datalab.plugins.recipes.**RecipeValidationError**

Raised when recipe inputs or parameters violate their descriptor.

Host-independent assignment of objects to recipe inputs, and readiness.

DataLab Desktop and DataLab-Web share this module: both propose bindings from the selected objects, assess whether a recipe can run on them, and phrase the returned issues in their own user interface.

datalab.plugins.recipe_binding.**INPUT_CHECK_FAILED_CODE = 'input-check-failed'**

Diagnostic code of an exception raised by a recipe input check

class datalab.plugins.recipe_binding.**RecipeBindingProposal**(bindings: Mapping[str, tuple[SignalObj | ImageObj, ...]], ambiguous_slots: tuple[str, ...] = ())

Objects proposed for each recipe input slot.

class datalab.plugins.recipe_binding.**RecipeInputIssue**(code: RecipeInputIssueCode, slot_id: str, details: Mapping[str, object]=<factory>)

Generic problem found on one recipe input slot.

`details` holds what hosts need to phrase the problem: `count` and `min_count` (`TOO_FEW`), `count` (`TOO_MANY`), `key`, `count` and `titles` of the objects lacking it (`MISSING_METADATA`), `count` and `titles` (`WRONG_TYPE`, `DUPLICATE`).

class datalab.plugins.recipe_binding.**RecipeInputIssueCode**(value)

Generic reason why objects cannot be bound to a recipe input slot.

class datalab.plugins.recipe_binding.**RecipeReadiness**(status: RecipeReadinessStatus, bindings: Mapping[str, tuple[SignalObj | ImageObj, ...]], issues: tuple[RecipeInputIssue, ...] = (), diagnostics: tuple[RecipeDiagnostic, ...] = ())

Result of the assessment of recipe inputs.

property can_run: bool

Return True if the recipe can run with the assessed bindings.

class datalab.plugins.recipe_binding.**RecipeReadinessStatus**(value)

Whether a recipe can run on a set of candidate objects.

NO_INPUT = 'no_input'

No candidate object has a type accepted by the recipe

NEEDS_ASSIGNMENT = 'needs_assignment'

Inputs are available but must be assigned to slots by the user

NOT_READY = 'not_ready'

Inputs violate the slot declarations or the recipe input checks

WARNINGS = 'warnings'

The recipe can run, but its input checks returned warnings

READY = 'ready'

The recipe can run

datalab.plugins.recipe_binding.**assess_recipe_inputs**(descriptor: RecipeDescriptor, candidates: Sequence[SignalObj | ImageObj] = (), parameters: DataSet | None = None, bindings: Mapping[str, Sequence[SignalObj | ImageObj]] | None = None) → RecipeReadiness

Assess whether a recipe can run on candidate objects or bindings.

Parameters

- **descriptor** – recipe descriptor
- **candidates** – candidate objects, used when `bindings` is None
- **parameters** – parameters passed to the recipe input checks
- **bindings** – explicit bindings (e.g. edited by the user)

Returns

Readiness status, bindings, generic issues, and recipe diagnostics

datalab.plugins.recipe_binding.**check_recipe_inputs**(descriptor: RecipeDescriptor, inputs: Mapping[str, tuple[SignalObj | ImageObj, ...]], parameters: DataSet | None) → tuple[RecipeDiagnostic, ...]

Run the input checks of a recipe on complete inputs.

An exception raised by the checks (e.g. by a validation function reused from the recipe) is reported as an error diagnostic.

Parameters

- **descriptor** – recipe descriptor
- **inputs** – objects bound to each slot
- **parameters** – recipe parameters, or None for a parameterless recipe

Returns

Diagnostics returned by the recipe

datalab.plugins.recipe_binding.**create_recipe_parameters**(descriptor: RecipeDescriptor, values: Mapping[str, object] | None = None) → DataSet | None

Create default recipe parameters, updated with known values.

Parameters

- **descriptor** – recipe descriptor
- **values** – parameter values, e.g. suited to an example

Returns
Parameters, or None for a parameterless recipe

Raises
RecipeValidationError – a value does not match a recipe parameter

datalab.plugins.recipe_binding.**find_input_issues**(descriptor: RecipeDescriptor, bindings: Mapping[str, Sequence[SignalObj | ImageObj]], ambiguous_slots: Sequence[str] = ()) → tuple[RecipeInputIssue, ...]

Check bindings against the declaration of the recipe input slots.

Parameters

- **descriptor** – recipe descriptor
- **bindings** – objects bound to each slot
- **ambiguous_slots** – unbound slots awaiting a user assignment

Returns
Problems found, in slot declaration order

datalab.plugins.recipe_binding.**is_compatible**(slot: RecipeInputSlot, obj: object) → bool

Return True if an object has the type accepted by an input slot.

datalab.plugins.recipe_binding.**propose_bindings**(descriptor: RecipeDescriptor, candidates: Sequence[SignalObj | ImageObj]) → RecipeBindingProposal

Propose objects for each input slot of a recipe.

The recipe's binding suggestions come first. Otherwise, all compatible candidates go to the only slot accepting their type; slots sharing a type with others are left to the user (ambiguous slots).

Parameters

- **descriptor** – recipe descriptor
- **candidates** – candidate objects, e.g. the current selection

Returns
Proposed bindings and the slots requiring a user decision

Portable declarations for plugin-provided example workspaces.

class datalab.plugins.examples.**PluginExample**(id: str, title: str, resource: str | None = None, description: str = '', recipe_ids: Sequence[str] = (), expected_checks: Sequence[str] = ())

Packaged or generated example, and the recipes it was designed for.

An example without recipe is presented as a plain dataset.

property package: str

Return the importable package containing the example resource.

property resource_path: str

Return the POSIX-style path relative to the resource package.

resolve() → Traversable

Resolve the declared resource without requiring a filesystem path.

as_file() → Iterator[Path]

Materialize the resource for APIs that require a filesystem path.

class datalab.plugins.examples.**PluginExampleData**(objects: ~collections.abc.Sequence[~sigima.objects.signal.object.SignalObj | ~sigima.objects.image.object.ImageObj], parameter_values: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, object]] = <factory>)

Scientific objects and per-recipe defaults generated by a plugin.

Parameters

- **objects** – generated signals and images
- **parameter_values** – recipe ID to parameter values suited to these objects

values_for(recipe_id: str) → Mapping[str, object]

Return the parameter values suited to one recipe (empty if none).

Portable declarations for plugin-provided welcome page tiles.

class datalab.plugins.tiles.**WelcomeTile**(id: str, title: str, description: str = '', icon: str | None = None, launcher: str | None = None)

Welcome page tile exposed by an application plugin

Parameters

- **id** – plugin-local tile ID (lowercase letters, digits, `.`, `_`, `-`)
- **title** – tile title
- **description** – short description shown below the title
- **icon** – `package:path` resource (SVG or bitmap) or DataLab icon file name; `None` falls back to the plugin icon
- **launcher** – name of the plugin method called when the tile is clicked; `None` shows the plugin page in the Applications panel

property is_package_icon: bool

Return True if the icon is a `package:path` resource

Declarations of plugin-owned tools listed in the Applications catalog.

DataLab Desktop and DataLab-Web share this module.

class datalab.plugins.tools.**PluginTool**(id: str, title: str, launcher: str | None = None, description: str = '', icon: str | None = None, instrument: str | None = None, object_type: RecipeObjectType | None = None, selection: ToolSelection = ToolSelection.NONE, menu: bool = True)

Tool with its own user interface, exposed by an application plugin

Recipes cover headless analyses run by DataLab; a tool covers any other interaction owned by the plugin (wizard, interactive editor, data preparation, instrument…). DataLab lists tools in the Applications catalog and in the plugin's submenu of the Plugins menu.

A tool is opened either by a `launcher` method, called without arguments, or by an `instrument` method returning a *`datalab.plugins.instruments.PluginInstrument`*, which DataLab shows in a window with a live view and the instrument settings.

Parameters

- **id** – plugin-local tool ID (lowercase letters, digits, `.`, `_`, `-`)
- **title** – tool title
- **launcher** – name of the plugin method called, without arguments, to open the tool
- **description** – short description of what the tool does
- **icon** – `package:path` resource (SVG or bitmap) or DataLab icon file name; `None` falls back to the plugin icon
- **instrument** – name of the plugin method returning the instrument shown by DataLab (exclusive with `launcher`)
- **object_type** – type of the objects the tool works on or creates; the tool is listed in the menu of that panel (`None`: both panels)
- **selection** – selection needed to open the tool, counting objects of `object_type` only
- **menu** – if False, DataLab does not add the tool to the plugin's submenu of the Plugins menu (e.g. because the plugin already places it in one of its own submenus); it is still listed in the Applications catalog

class datalab.plugins.tools.**ToolSelection**(value)

Selection a tool needs to be opened (names follow DataLab's conditions).

datalab.plugins.tools.**tool_accepts_selection**(tool: PluginTool, objects: Sequence[SignalObj | ImageObj]) → bool

Return True if the selected objects allow opening the tool.

Parameters

- **tool** – tool to open
- **objects** – selected objects

Returns

True if the number of selected objects of the tool's object type matches the tool's selection

Instruments exposed by application plugins as tools.

An instrument is a headless object: it holds settings (a guidata DataSet), returns a live frame for these settings, and acquires objects. DataLab shows it in a window with the live view on the left and the settings on the right. DataLab Desktop and DataLab-Web share this module.

class datalab.plugins.instruments.**InstrumentAcquisition**(group_title: str, objects: Sequence[SignalObj | ImageObj])

Objects acquired by an instrument, added by DataLab to the workspace.

DataLab adds them to a new group, or to the current group of the panel when the user turns off the New group option of the instrument window (see *PluginInstrument.new_group*).

Parameters

- **group_title** – title of the group created for the acquisition, if any
- **objects** – acquired signals, or acquired images

class datalab.plugins.instruments.**InstrumentFrame**(objects: Sequence[SignalObj | ImageObj], summary: str = '', value_range: tuple[float, float] | None = None)

Live frame shown by DataLab for the current instrument settings.

Parameters

- **objects** – signals drawn together, or a single image
- **summary** – short text shown below the view (e.g. measured levels)
- **value_range** – fixed Y range of signals, or color range of the image (`None`: automatic)

class datalab.plugins.instruments.**PluginInstrument**(settings: DataSet)

Instrument opened by a plugin tool.

DataLab writes the edited values into `settings` before calling *`preview()`* or *`acquire()`*. Both methods raise `ValueError` with a user-facing message when the settings cannot be used. The plugin keeps one instance per tool during the session, so settings are kept from one opening to the next.

DataLab calls *`open()`* when the instrument window opens and *`close()`* when it closes (including when the plugin is reloaded or DataLab quits). It calls these methods, *`preview()`* and *`acquire()`* one at a time, outside the GUI thread, even when the window is closed and opened again while a call is running, and writes `settings` only between two calls: a slow instrument does not freeze DataLab. Data received by the instrument in its own threads (e.g. device events) must be stored for the next *`preview()`* or *`acquire()`* call.

Parameters

settings – instrument settings, edited by DataLab

live_interval_ms: int = 500

Interval between two frames in live mode, in milliseconds

new_group: bool = True

Default state of the New group option of the instrument window (if True, each acquisition is added to a new group, otherwise to the current group of the panel). The choice of the user, once made, prevails.

open() → None

Prepare the instrument when its window opens (e.g. connect to it).

Raises `ValueError` with a user-facing message if the instrument cannot be used: DataLab shows this message, and does not call *`preview()`* or *`acquire()`* until a new request (e.g. after a settings change) opens the instrument successfully. Does nothing by default.

close() → None

Release the instrument when its window closes (e.g. disconnect).

Called even if *`open()`* failed, and possibly several times: it must release only what is still held. Does nothing by default.

abstractmethod preview() → InstrumentFrame

Return a live frame for the current settings.

abstractmethod acquire() → InstrumentAcquisition

Acquire objects with the current settings.

Package resources declared by plugins with the `package:path` syntax.

datalab.plugins.resources.**LOCAL_ID_PATTERN = re.compile('^[a-z0-9]+(?:[._-][a-z0-9]+)*$')**

Plugin-local identifier syntax (examples, welcome page tiles)

datalab.plugins.resources.**resolve_package_resource**(resource: str, label: str = 'Plugin resource') → Traversable

Resolve a `package:path` resource without requiring a filesystem path

Parameters

- **resource** – resource declaration, e.g. `"my_plugin:icons/logo.svg"`
- **label** – resource description used in error messages

Returns
`importlib.resources` traversable, also valid for zipped packages

Raises

- **ValueError** – if the declaration is malformed
- **FileNotFoundError** – if the package does not contain the resource

`datalab.plugins.resources.`**split_package_resource**(resource: str, label: str = 'Plugin resource') → tuple[str, str]

Validate a `package:path` resource and return its two parts

Parameters

- **resource** – resource declaration, e.g. `"my_plugin:icons/logo.svg"`
- **label** – resource description used in error messages

Returns
Importable package name and POSIX path relative to this package

Raises
ValueError – if the declaration is malformed

Transactional Desktop runner for plugin-provided scientific recipes.

exception `datalab.gui.plugins.recipe_runner.`**RecipeCommitError**(message: str, rollback_error: Exception | None = None)

Raised after a failed recipe workspace commit has been rolled back.

rollback_error
Exception raised by the rollback itself, or None when the workspace was successfully restored.

class `datalab.gui.plugins.recipe_runner.`**RecipeRunner**(mainwindow: DLMainWindow)

Validate, execute, and transactionally commit a recipe on Desktop.

run(descriptor: RecipeDescriptor, inputs: Mapping[str, Sequence[SignalObj | ImageObj]], parameters: DataSet | None = None, context: RecipeExecutionContext | None = None, group_title: str | None = None) → RecipeOutcome

Validate, execute, and atomically commit one recipe invocation.

3.5.2 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
  - Plugin file naming convention
  - New result objects
  - Object UUID access
  - 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
```

Plugin file naming convention

Plugin filenames must follow the new naming convention:

> **Important**
>
> Plugin files must be renamed from `cdl_*.py` to `datalab_*.py`.

Examples:

- `cdl_myplugin.py` → `datalab_myplugin.py`
- `cdl_custom_filters.py` → `datalab_custom_filters.py`
- `cdl_example.py` → `datalab_example.py`

This naming convention is required for DataLab to automatically discover and load plugins at startup. Plugins that don't follow this naming pattern will not be loaded.

> **Note**
>
> The plugin discovery mechanism looks for Python files matching the pattern `datalab_*.py` in the plugins directory. Make sure to update your plugin filenames accordingly.

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`).

Object UUID access

The way to access an object's UUID has changed:

```
# v0.20 - Direct attribute access
obj_uuid = obj.uuid

# v1.0 - Use helper function from datalab.objectmodel
from datalab.objectmodel import get_uuid
obj_uuid = get_uuid(obj)

# Alternative in v1.0 - Access metadata directly
obj_uuid = obj.get_metadata_option("uuid")
```

This change applies to both `SignalObj` and `ImageObj` objects. The UUID is now stored in the object's metadata rather than as a direct attribute.

> **Note**
>
> The `get_uuid()` function is defined in `datalab.objectmodel`, not as a method on the object itself. Import it before use: `from datalab.objectmodel import get_uuid`.

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 8: 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.base.dst_1_to_1()`
`cdl.computation.signal.dst_11()`	`sigima.proc.base.dst_1_to_1()`
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 8 – 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_signa`	`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=`	`edit_new_image_parameters(hide_dtype=True)`
`edit_new_image_parameters(hide_image_type=T`	`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.read_image()`
`cdl.io.image.write_image()`	`sigima.io.write_image()`
`cdl.io.signal.read_signal()`	`sigima.io.read_signal()`
`cdl.io.signal.write_signal()`	`sigima.io.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]
        differences = np.abs(values - central_pixel)
        mean_difference = np.mean(differences)
```

(continues on next page)

(continued from previous page)

```
        if mean_difference == 0:
            return central_pixel
        weights = np.exp(-differences / mean_difference)
        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)
        mean_difference = np.mean(differences)
        if mean_difference == 0:
            return central_pixel
        weights = np.exp(-differences / mean_difference)
        return np.average(values, weights=weights)
    return spi.generic_filter(data, filter_func, size=5)


class CustomFilters(datalab.plugins.PluginBase):
```

(continues on next page)

(continued from previous page)

```
    """DataLab Custom Filters Plugin"""

    PLUGIN_INFO = datalab.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 = 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)
```

(continues on next page)

(continued from previous page)

```
        panel.processor.compute_binning(param)
        panel.processor.compute_moving_median(cdl.param.MovingMedianParam.create(n=5))

    def detect_blobs(self) -> None:
        """Detect circular blobs"""
        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()
```

(continues on next page)

(continued from previous page)

```
        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.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"),
    )
```

(continues on next page)

(continued from previous page)

```
    def add_noise_to_image(self) -> None:
        """Add noise to image"""
        self.imagepanel.processor.compute_11(
            add_noise_to_image,
            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


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_1_to_1(
            sipi.add_gaussian_noise,
            paramclass=sigima.objects.NormalDistributionParam,
```

(continues on next page)

(continued from previous page)

```
            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. The custom `add_noise_to_image` function, built with `dst_11` (now `sigima.proc.base.dst_1_to_1`) and a test data helper, is replaced by `sigima.proc.image.add_gaussian_noise`
5. `compute_11` → `compute_1_to_1`
6. `dlo.NormalRandomParam` → `sigima.objects.NormalDistributionParam`
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
```

(continues on next page)

(continued from previous page)

```
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)
if geometry is None:
    raise ValueError("The signal has no suitable bandwidth crossings")

# Access segment coordinates [x0, y0, x1, y1]
x0, y0, x1, y1 = geometry.coords[0]
print(f"Bandwidth segment: ({x0}, {y0}) to ({x1}, {y1})")

# Calculate bandwidth length
bandwidth = geometry.segments_lengths()[0]
print(f"Bandwidth@-3dB: {bandwidth:.2f}")
```

(continues on next page)

(continued from previous page)

```
# 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
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_binning(param)  # image processor (or run_feature("binning", param))

# Peak detection (signal): ``compute_peak_detection`` still exists and
# opens the interactive threshold dialog as before, but it now stores
# the result as an XY-markers table (no sticks signal is created
# automatically). To rebuild the legacy sticks signal, chain with
# ``compute_markers_to_signal`` (or call ``run_feature("extract_peak_positions",
# param)`` directly when no interactive dialog is needed):
proc.compute_peak_detection()             # interactive, XY-markers result
proc.compute_markers_to_signal()          # optional: rebuild sticks signal
```

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_moving_median'`

Solution: Replace `compute_*` method calls with `run_feature()` calls:

```
# Wrong (v0.20 style)
proc.compute_moving_median(param)

# Correct (v1.0 style)
proc.run_feature("moving_median", 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/builtin` directory

3.5.3 Macros

Overview

There are many ways to extend DataLab with new functionality (see Plugins or Remote controlling (XML-RPC)). 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.



Fig. 74: 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 (XML-RPC) section.

Main features

The Macro Panel is a simple interface to:

- Create new macros, using the "New macro"  button. The button also exposes a dropdown menu offering a blank macro and the bundled templates (see Templates).
- Rename existing macros, using the "Rename macro"  button (or by double-clicking on a tab).
- Duplicate the current macro, using the "Duplicate macro"  button.
- Import/export macros from/to files, using the "Import macro"  and "Export macro"  buttons.
- Reopen a macro from previous sessions, using the "Recent macros…"  button (the recent list is persisted across DataLab sessions).
- Execute macros, using the "Run macro"  button.
- Stop the execution of a macro, using the "Stop macro"  button.

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.

Editor features

The macro editor offers a few features that make script writing more comfortable:

- Autosave: every edit is saved, after a short delay, to a recovery file of the DataLab configuration folder (independent of the workspace), so unsaved edits survive an unexpected shutdown. When the Macro panel is shown after a restart, DataLab offers to restore the macros found in this file.
- Tab persistence: the set of open macros and the active tab are restored the next time DataLab starts, so you pick up exactly where you left off.
- Find / Replace (`Ctrl+F` / `Ctrl+H`): an inline bar at the bottom of the editor supports case-sensitive search, whole-word matching, and replace (or replace-all) inside the active macro.
- Python autocompletion is enabled out of the box (`Ctrl+Space` to trigger it manually), backed by the embedded code editor.

Console

The bottom Console panel captures the standard output and standard error streams produced by the running macro. Right-clicking inside the console exposes:

- Clear console — wipe the current contents.
- Save history log… — export the console history to a text file.

The maximum number of lines retained by the console can be tuned via the `console_max_lines` option of the `[macro]` section of the configuration file (`Conf.macro_console_max_lines`, defaults to `5000`).

Templates

The "New macro" dropdown lists a set of ready-to-use templates covering common patterns (signal creation, image processing, batch loops, etc.). You can ship your own templates by dropping `*.py` files into the user templates directory — by default `~/.DataLab_v1/macro_templates/`, configurable via the `templates_path` option of the `[macro]` section of the configuration file (`Conf.macro_templates_path`). The menu entry title of a template is read from its first line when it starts with `# DataLab template:` (e.g. `# DataLab template: Batch processing of images`); otherwise, it is derived from the file name. Bundled templates take precedence over user templates when names collide.

Example

For a detailed example of how to create a macro, see the Prototyping a custom processing pipeline tutorial.

3.5.4 AI Assistant

DataLab embeds an optional AI Assistant: a dockable chat panel connected to a Large Language Model (LLM) that can inspect your workspace, create and process signals and images, and write and run Python macros on your behalf.

> **Warning**
>
> The AI Assistant is optional. When you choose a remote provider, it receives your messages and any workspace information you ask it to use. You can instead choose a local model to keep this processing on your machine. Review the Privacy and security section before using a remote provider on sensitive data.

Getting started

1. Open `File > Settings` and go to the AI Assistant tab, or click the Settings… button of the AI Assistant panel (see AI Assistant for the complete list of options).
2. Choose a provider and a model, then set the API key — or, better, leave the field empty and export the provider's standard environment variable.
3. Click Test connection to check the endpoint before starting.
4. Show the AI Assistant panel from the `View` menu if it is hidden. It is docked on the right side of the main window, tabified with the Macro panel.

> **Tip**
>
> To discover the assistant without any account or API key, select the `mock` provider: it replies with scripted answers triggered by simple keywords, and exercises the whole pipeline (tool calls, confirmation dialogs, conversation storage) offline.

Supported providers

The assistant talks to any service exposing an OpenAI-compatible chat completion API. This covers, among others:

- OpenAI (`https://api.openai.com/v1`)
- GitHub Models (`https://models.github.ai/inference`)
- Azure OpenAI
- Local runtimes such as Ollama, LM Studio, llama.cpp or vLLM

The Load preset… button of the settings dialog pre-fills the Base URL and Model fields for the most common endpoints.

Using the chat panel

The panel is made of a toolbar, the conversation view and an input box:

- New conversation: start a fresh conversation. The previous one is kept in the conversation store.
- History…: browse past conversations, load, rename or delete them, and export one as a Markdown file. Conversations are stored in the DataLab user configuration directory.
- Settings…: open the AI Assistant tab of the settings dialog (provider, model, API key…).
- Send / Stop: send the current message, or interrupt an ongoing request. Requests run in a background thread, so the interface stays responsive.
- The token counter on the right of the toolbar shows the context size of the last request and the cumulated usage of the conversation.

The input box keeps a history of previously sent messages, navigable with the keyboard like a shell prompt.

When the assistant proposes a macro, the generated code is transient: it is executed without cluttering the Macro panel. A Save to Macros link is offered after the run if you want to keep it.

When you ask the assistant to write a macro rather than to run it, the code is only proposed: the confirmation dialog states that the macro will not be executed, and the same Save to Macros link is offered. Such a link is also offered under each Python code block of the assistant's replies (unless macro creation is disabled), and all these links are available again when a conversation is reopened from History…. Once a macro is saved, a Show in Macro panel link brings that panel to the front.

Tools available to the assistant

The assistant does not act directly on DataLab: it may only call a fixed set of declared tools. Tools marked as read-only never modify the workspace and may be auto-approved (see the corresponding setting); every other tool requires an explicit confirmation.

Tool	Read-only	Purpose
`list_objects`	✓	List the signals or images of a panel
`get_current_panel`	✓	Return the currently active panel
`get_object_info`	✓	Inspect a specific object (shape, units, metadata, …)
`list_available_operations`	✓	Introspect the processing catalog exposed by Sigima
`list_plugin_actions`	✓	List the actions contributed by third-party plugins
`get_macro_console_output`	✓	Read back the Macro panel console
`get_api_help`	✓	Return the public API of the proxy, `SignalObj`, `ImageObj` or `sigima.params`, so the model does not invent method names
`capture_view`	✓	Grab a screenshot of the current plot and inject it in the conversation, so a multimodal model can visually inspect the data
`trigger_plugin_action`		Trigger a plugin action by its menu path
`create_synthetic_signal`		Create a synthetic signal (sine, cosine, Gaussian, noise, ramp)
`create_synthetic_image`		Create a synthetic image (2D Gaussian, ramp, noise, checkerboard)
`load_file`		Load a file into a panel
`apply_operation`		Run any registered processing feature
`create_and_run_macro`		Create and execute a Python macro, or only propose it for saving

> **Note**
>
> The `create_and_run_macro` tool is only exposed to the model when the Allow AI to create and run macros option is enabled. When it is disabled, the assistant cannot even propose arbitrary code execution.

Confirming tool calls

Whenever the assistant wants to run a tool that modifies the workspace, a confirmation dialog shows the tool name and the parameter values. When a macro is proposed, the dialog also displays its syntax-highlighted source code, so that the code can be reviewed before being executed. Rejecting the call returns the refusal to the model, which may then propose something else.

Privacy and security

- No request without your action: the assistant contacts its provider only when you send a message in the panel, or when you click Test connection in the settings. Opening the panel or DataLab sends nothing. Note that with the `openai` provider, a key found in the `OPENAI_API_KEY` environment variable is used even if the API key field is empty.
- No telemetry: DataLab does not log or forward the content of your conversations anywhere else than to the provider you configured.
- Local providers: to keep everything on your machine, point the Base URL to a local runtime (Ollama, LM Studio, llama.cpp, vLLM). Data then never leaves your computer.
- API key storage: the key is stored in plain text in the DataLab configuration file. Prefer the provider's environment variable.
- Explicit confirmation: every action modifying the workspace — and every macro execution — requires an explicit user confirmation by default.

- Iteration cap: the number of chained tool calls is bounded by the Max tool-call iterations setting, so a misbehaving model cannot loop indefinitely.

See also

The macro system used by the assistant is described in Macros, and the underlying control API in Remote controlling (XML-RPC).

3.5.5 Web API

The DataLab Web API provides a modern HTTP/JSON interface for external access to the DataLab workspace. It is the recommended integration path for new projects, especially for Jupyter notebooks and web-based tools.

Note

The Web API is available starting from DataLab 1.1. The required dependencies (FastAPI, Uvicorn, Pydantic) are included in the standard DataLab installation.

Overview

The Web API enables:

- Jupyter notebook integration via DataLab-Kernel
- WASM/Pyodide compatibility (no XML-RPC dependency)
- External tool integration (any HTTP client)

Unlike the XML-RPC interface, the Web API:

- Uses JSON for metadata and NPZ for binary data (efficient large array transfer)
- Provides bearer token authentication for security
- Follows REST conventions with OpenAPI documentation
- Works in browser environments (WASM/Pyodide)

Table 9: Comparison with XML-RPC

Feature	XML-RPC	Web API
Protocol	XML	JSON + Binary (NPZ)
WASM Support	×	✓
Large Arrays	Slow (base64)	Fast (NPZ)
Authentication	None	Bearer Token
Standards	XML-RPC	REST + OpenAPI

Quick Start

Enabling the Web API

There are several ways to enable the Web API server:

1. Via UI: File → Web API → Start Web API Server
2. Via environment variable: Set `DATALAB_WEBAPI_ENABLED=1` before starting DataLab

When started from the menu or from the status bar, DataLab displays the server URL and authentication token in a dialog. The status bar also shows the Web API port when the server is running: click it to display the connection information again.

The default port is 18080 (or the next available port if busy).

Auto-Discovery

DataLab-Kernel can automatically discover and connect to a running DataLab instance without any manual configuration. When the Web API starts, DataLab writes connection information to a file that DataLab-Kernel reads automatically.

Just load the extension — no environment variables or explicit connection needed:

```
# In your notebook (JupyterLab, VS Code, etc.)
%load_ext datalab_kernel

# DataLab-Kernel automatically finds and connects to DataLab
workspace.list()  # Already connected!
```

The auto-discovery mechanism tries the following methods in order:

1. Environment variables (`DATALAB_WORKSPACE_URL`, `DATALAB_WORKSPACE_TOKEN`)
2. Connection file written by DataLab (native Python only)
3. URL query parameters (JupyterLite: `?datalab_url=...&datalab_token=...`)
4. Well-known port probing (`http://127.0.0.1:18080`)

If discovery fails, DataLab-Kernel starts in standalone mode and you can connect later using `workspace.connect()`.

Manual Connection (Legacy)

If auto-discovery doesn't work (e.g., running on a different machine), you can set environment variables before starting your notebook kernel:

```
export DATALAB_WORKSPACE_URL=http://127.0.0.1:18080
export DATALAB_WORKSPACE_TOKEN=<your-token>
```

Then, in a notebook using the DataLab-Kernel kernel (or after `%load_ext datalab_kernel`), use the `workspace` object which is already defined in the namespace:

```
# List objects in DataLab
workspace.list()

# Retrieve an object
signal = workspace.get("my_signal")

# Add a processed result back to DataLab
workspace.add("processed", result)
```

API Reference

Base URL: `http://127.0.0.1:<port>/api/v1`

Status Endpoint (No Auth Required)

Method	Endpoint	Description
GET	`/status`	Server status and version information

Object Operations (Auth Required)

All endpoints below require the `Authorization: Bearer <token>` header, except for requests from the local machine while the localhost bypass is enabled (default, see Localhost Token Bypass).

Method	Endpoint	Description
GET	`/objects`	List all objects with metadata (JSON)
GET	`/objects/{name}`	Get object metadata (JSON)
DELETE	`/objects/{name}`	Delete an object
PATCH	`/objects/{name}/metadata`	Update object metadata

Binary Data Transfer (Auth Required)

Methoc	Endpoint	Description
GET	`/objects/{name}/data`	Download object as NPZ archive (supports `?compress=false`)
PUT	`/objects/{name}/data`	Create or replace an object from an NPZ archive (supports `?overwrite=true`)
PUT	`/objects/{name}`	Update an existing object in-place from an NPZ archive, preserving its identity, group membership and position

The `GET /objects/{name}/data` endpoint streams an NPZ archive as `application/x-npz`. Pass `?compress=false` to skip ZIP deflate compression when the client (e.g. a notebook) prefers a faster transfer at the cost of larger payloads. Object names containing non-ASCII characters are returned in the `Content-Disposition` header using the RFC 5987 `filename*` form.

The `PUT /objects/{name}/data` endpoint creates a new object with the given name, or returns `409 Conflict` if it already exists. Pass `?overwrite=true` to replace the existing object: the new object is added first, then the existing one is removed, so that the existing object is kept if the new one cannot be added. While the replacement is in progress, the workspace briefly contains both objects.

Computation (Auth Required)

Method	Endpoint	Description
POST	`/select`	Select objects in a panel for subsequent computations
POST	`/calc`	Run a DataLab computation function on the current selection

Data Format

Objects are transferred using NumPy's NPZ format for efficiency. The format is implemented by Sigima (`sigima.io.serialize_object_to_npz` and `sigima.io.deserialize_object_from_npz`), so that Python clients can read and write these archives with the same code as DataLab.

SignalObj NPZ Structure

```
my_signal.npz
├── x.npy          # X coordinates (float64)
├── y.npy          # Y data (float64)
├── dx.npy         # X uncertainties (optional)
├── dy.npy         # Y uncertainties (optional)
└── metadata.json  # Type ("signal"), title, labels, units,
                   # object metadata (optional)
```

ImageObj NPZ Structure

```
my_image.npz
├── data.npy       # 2D image array (preserves dtype)
├── xcoords.npy    # X pixel centers (only for non-uniform coordinates)
├── ycoords.npy    # Y pixel centers (only for non-uniform coordinates)
└── metadata.json  # Type ("image"), title, labels, units,
                   # coordinates (x0, y0, dx, dy, is_uniform_coords),
                   # object metadata (optional)
```

In `metadata.json`, the `type` key (`"signal"` or `"image"`) is required to read an archive. The optional `obj_metadata` key holds the metadata of the object (e.g. analysis results).

Authentication

All API endpoints except `/status` require a Bearer token, unless the request comes from the local machine while the localhost bypass is enabled (default, see Localhost Token Bypass):

```
Authorization: Bearer <token>
```

The token is generated when the server starts and displayed in the connection dialog.

Environment Variables

Variable	Description	Default
`DATALAB_WEBAPI_HOST`	Bind address	`127.0.0.1`
`DATALAB_WEBAPI_PORT`	Server port	`18080` (or next available)
`DATALAB_WEBAPI_TOKEN`	Auth token	Generated

Security Model

The Web API implements the following security measures:

1. Localhost binding: By default, the server only accepts connections from the local machine (127.0.0.1).
2. Token authentication: Every request (except status) must include a valid Bearer token, except for requests from the local machine while the localhost bypass is enabled, which is the default (see Localhost Token Bypass). With the default localhost binding, local clients therefore connect without a token until you disable this bypass.
3. Explicit opt-in: Remote binding (0.0.0.0) requires explicit configuration.

Localhost Token Bypass

By default, DataLab does not check the token of requests coming from the local machine (`127.0.0.1` or `::1`): local clients, such as DataLab-Kernel, connect without a token. This makes auto-discovery work seamlessly even when the connection file cannot be read (e.g., in JupyterLite or sandboxed environments).

To require the token for local requests too, disable the Web API localhost bypass option in the General tab of File → Settings…. The change applies the next time the Web API server starts.

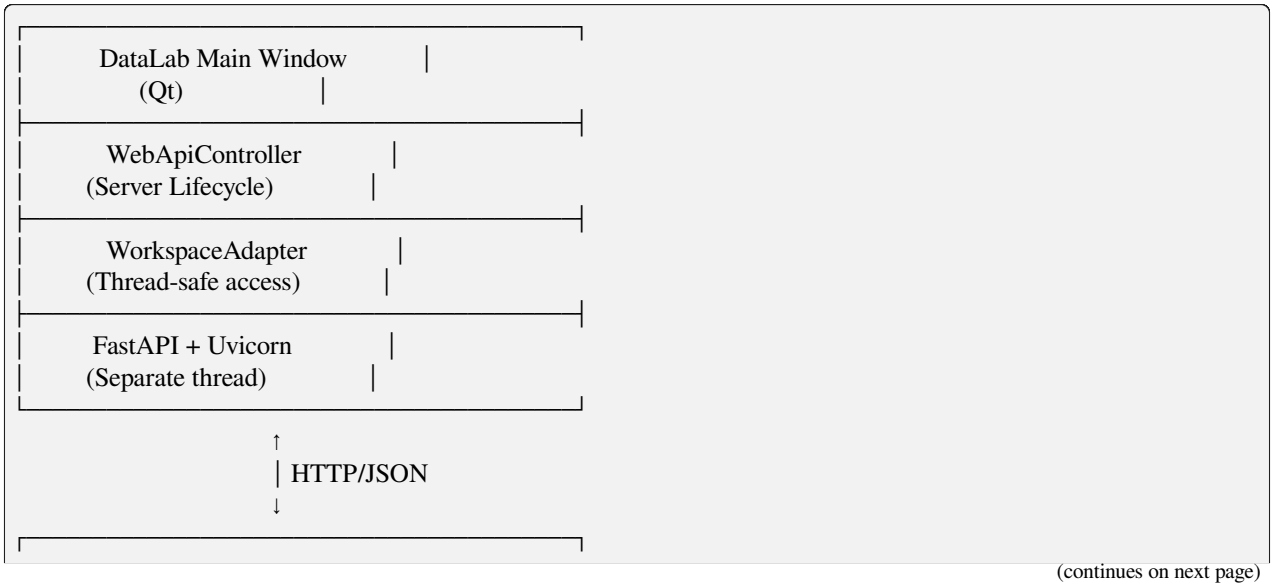
Warning

While the Web API server is running with the localhost bypass enabled (default), any software running on your machine, including malicious software, and any web page open in your browser can access your data without a token. Disable the bypass if you do not control all applications running on your machine or if you browse untrusted websites while the server is running.

Warning

Exposing the API to the network allows anyone with the token to access your data. Use with caution on trusted networks only.

Architecture



(continues on next page)

(continued from previous page)

```
│     DataLab-Kernel / Client          │
└──────────────────────────────────────┘
```

The Web API runs in a separate thread from the Qt GUI. The `WorkspaceAdapter` ensures thread-safe access to DataLab objects by marshalling calls to the main thread using Qt's signal/slot mechanism.

Installation

The Web API dependencies are included in the standard DataLab installation:

- FastAPI (web framework)
- Uvicorn (ASGI server)
- Pydantic (data validation)

No additional installation steps are required.

Programmatic Control

The Web API can also be started and controlled programmatically via the XML-RPC interface, which is useful for automated workflows:

```python
from datalab.control.remote import RemoteClient

proxy = RemoteClient()
proxy.connect()

# Start the Web API server
info = proxy.start_webapi_server()
print(f"URL: {info['url']}")
print(f"Token: {info['token']}")

# Check status
status = proxy.get_webapi_status()
print(f"Running: {status['running']}")

# Stop the server
proxy.stop_webapi_server()
```

See Also

- Remote controlling (XML-RPC) for the XML-RPC interface (legacy)
- DataLab-Kernel documentation for Jupyter integration

3.5.6 Remote controlling (XML-RPC)

> **Note**
>
> For new projects, we recommend using the Web API instead of XML-RPC. The Web API provides better performance for large arrays, token-based authentication, and compatibility with browser environments (WASM/Pyodide).
>
> XML-RPC remains available for backward compatibility with existing integrations.

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 lightweight alternative to remote control DataLab (i.e. without installing the whole DataLab package and its dependencies in your environment), use the `sigima.client.SimpleRemoteProxy` class provided by the Sigima package (`pip install sigima`). It exposes most of the XML-RPC entry points of the `RemoteProxy` class described below, with a minimal set of dependencies: only `call_method`, `remove_object`, `export_capsule`, `open_capsule`, `start_webapi_server`, `stop_webapi_server` and `get_webapi_status` are missing.
>
> This class supersedes `cdlclient.SimpleRemoteProxy`: the `cdlclient` package is deprecated and archived since DataLab 1.3. Its connection dialog is now provided by SigimaX (see Connection dialog below).

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 must run on the same computer as DataLab: the XML-RPC server only accepts connections from the local machine (`127.0.0.1`). To control DataLab from another computer, forward the XML-RPC port through a secure channel, such as an SSH tunnel.

The remote client application may be run before or after DataLab. In the former case, the remote client application must try to connect to DataLab until it succeeds.

Supported features

Supported features are the following:

- Get DataLab version, raise or close its main window
- Switch to signal or image panel
- Toggle auto-refresh and the display of object titles
- Remove all signals and images, or the current object
- Save current session to a HDF5 file
- Open HDF5 files into current session
- Browse HDF5 file

- Export or open a workspace capsule
- Open signals or images from files or from a directory
- Add a signal, an image, an object or a group, or replace an object
- Get object list, UUIDs, groups, current and selected objects
- Select objects or groups
- Delete metadata, add annotations or a label with object title, get the shapes of an object
- Run calculation with parameters
- Import, run or stop a macro
- Call any public method of the main window or of a panel
- Start, stop or query the Web API server

See the Public API section below for the description of the corresponding methods.

> **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`

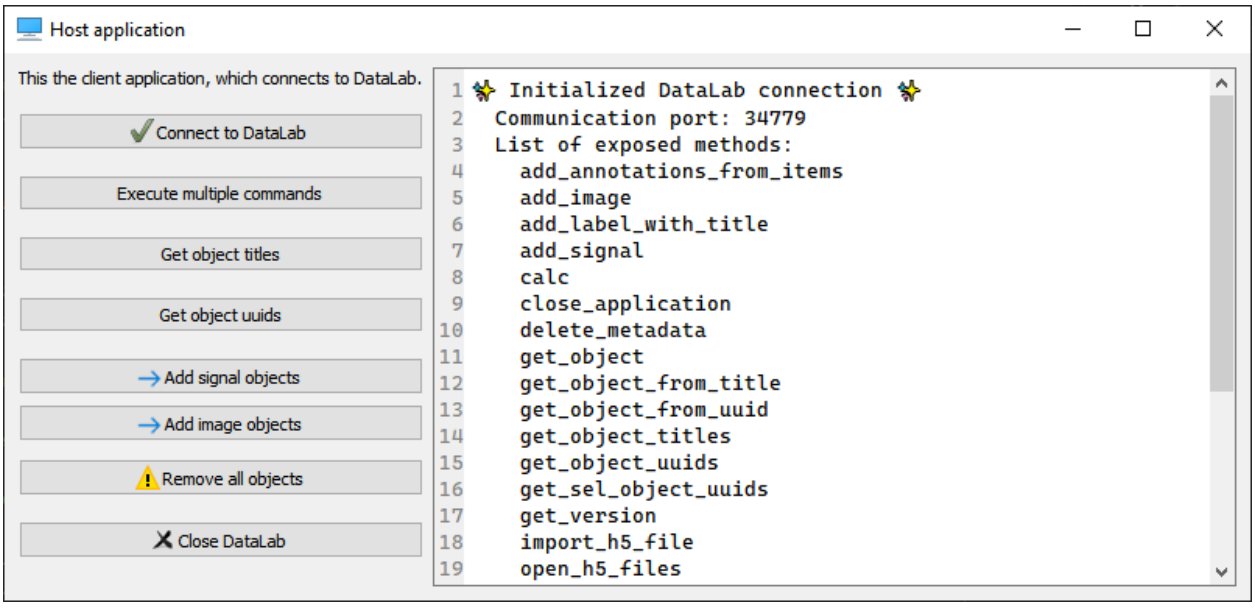


Fig. 75: Screenshot of remote client application test (`datalab.tests.features.control.remoteclient_app_test`)

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_v1", "DataLab_v1_typed.ini")
    ini = cp.ConfigParser()
    ini.read(fname)
    try:
        return ini.get("main", "rpc_server_port")
    except (cp.NoSectionError, cp.NoOptionError):
        raise socket.error("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 socket.error("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,
```

(continues on next page)

(continued from previous page)

```
            group_id,
            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()
```

Its automatic port discovery uses the configuration location of the Python-2-compatible guidata version. On macOS, or on Linux with a non-default `XDG_CONFIG_HOME`, that location can differ from the current DataLab configuration directory. In that case, read the port shown next to XML-RPC in DataLab's status bar and pass it explicitly as a string to `client.connect(port="28867")`, replacing `28867` with the actual port.

Connection dialog

A connection dialog may be used to connect to a running DataLab instance. It is exposed by the `sigimax.widgets.connection.ConnectionDialog` class, and takes optional `icon` and `banner` arguments to carry the branding of the client application. It works with any proxy exposing a blocking `connect()` method, i.e. both `RemoteProxy` and `sigima.client.SimpleRemoteProxy`.

Example of use:

```
# Copyright (c) DataLab Platform Developers, BSD 3-Clause license, see LICENSE file.

"""
DataLab Remote client connection dialog example
"""

# guitest: show,skip

from guidata.configtools import get_icon, get_image_file_path
from guidata.qthelpers import qt_app_context
from qtpy import QtGui as QG
```

(continues on next page)

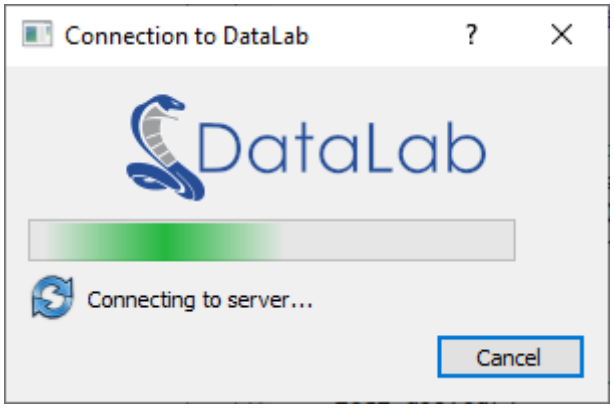


Fig. 76: Screenshot of connection dialog (`sigimax.widgets.connection.ConnectionDialog`)

(continued from previous page)

```
from qtpy import QtWidgets as QW
from sigimax.widgets.connection import ConnectionDialog

from datalab.control.proxy import RemoteProxy


def test_dialog():
    """Test connection dialog"""
    proxy = RemoteProxy(autoconnect=False)
    with qt_app_context():
        dlg = ConnectionDialog(
            proxy.connect,
            icon=get_icon("DataLab.svg"),
            banner=QG.QPixmap(get_image_file_path("DataLab-Banner-200.png")),
        )
        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.control.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]
<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

set_object(obj: SignalObj | ImageObj) → None

Set object data in DataLab.

Update an existing object in DataLab with new data from `obj`. The object is identified by its UUID (which is carried by `obj` from a previous *`get_object()`* call).

Parameters
obj – Signal or image object (must have the same UUID as an existing object in DataLab)

Raises
KeyError – if no object with matching UUID is found

load_from_files(filenames: list[str], param: DataSet | None = None) → None

Open objects from files in current panel (signals/images).

Parameters

- **filenames** – list of file names
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

load_from_directory(path: str, param: DataSet | None = None) → None

Open objects from directory in current panel (signals/images).

Parameters

- **path** – directory path
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

calc(name: str, param: 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.

call_method(method_name: str, *args, panel: str | None = None, **kwargs)

Call a public method on a panel or main window.

Method resolution order when panel is None: 1. Try main window (DLMainWindow) 2. If not found, try current panel (BaseDataPanel)

Parameters

- **method_name** – Name of the method to call
- ***args** – Positional arguments to pass to the method
- **panel** – Panel name ("signal", "image", or None for auto-detection). Defaults to None.
- ****kwargs** – Keyword arguments to pass to the method

Returns
The return value of the called method

Raises

- **AttributeError** – If the method does not exist or is not public
- **ValueError** – If the panel name is invalid

start_webapi_server(host: str | None = None, port: int | None = None) → dict

Start the Web API server.

Parameters

- **host** – Host address to bind to. Defaults to "127.0.0.1".
- **port** – Port number. Defaults to auto-detect available port.

Returns
Dictionary with "url" and "token" keys.

Raises
RuntimeError – If Web API dependencies not installed or server already running.

stop_webapi_server() → None

Stop the Web API server.

get_webapi_status() → dict

Get Web API server status.

Returns
Dictionary with "running", "url", "token" and "available" keys.

add_group(title: str, panel: str | None = None, select: bool = False) → str

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.

Returns
UUID of the created group

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.

export_capsule(filename: str, name: str | None = None, description: str | None = None, as_revision: bool = True, workspace_ref: str | None = None, workspace_filename: str | None = None, attachments: list[str] | None = None) → str

Export the workspace as a capsule (workspace and its provenance).

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **name** – Capsule name.
- **description** – Capsule description.
- **as_revision** – When the workspace was opened from a capsule, name that capsule as the parent revision. Defaults to True.
- **workspace_ref** – Absolute IRI where the workspace will be published: the capsule then references the workspace instead of including it.
- **workspace_filename** – File name of the workspace published at workspace_ref (default: last segment of the IRI, next to the capsule).
- **attachments** – Light files declared in the capsule (16 MiB at most each), never opened or executed.

Returns
Manifest fingerprint of the capsule.

get_current_object_uuid() → str | None

Return current object uuid in current panel.

Returns
UUID of the current object, or None if no object is current.

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_h5_workspace(h5files: list[str], reset_all: bool = False) → None

Load native DataLab HDF5 workspace files without any GUI elements.

This method can be safely called from scripts (e.g., internal console, macros) as it does not create any Qt widgets, dialogs, or progress bars.

> **Warning**
>
> This method only supports native DataLab HDF5 files. For importing arbitrary HDF5 files (non-native), use *open_h5_files()* or *import_h5_file()* instead.

Parameters

- **h5files** – List of native DataLab HDF5 filenames
- **reset_all** – Reset all application data before importing. Defaults to False.

Raises
ValueError – If a file is not a valid native DataLab HDF5 file

open_capsule(filename: str, workspace: str | None = None) → None

Open a capsule as the workspace, replacing the current one.

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **workspace** – Workspace file of a capsule that references its workspace instead of including it.

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

remove_object(force: bool = False) → None

Remove current object from current panel.

Parameters
force – if True, remove object without confirmation. Defaults to False.

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_h5_workspace(filename: str) → None

Save current workspace to a native DataLab HDF5 file without GUI elements.

This method can be safely called from scripts (e.g., internal console, macros) as it does not create any Qt widgets, dialogs, or progress bars.

Parameters
filename – HDF5 filename to save to

Raises
IOError – If file cannot be saved

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

3.5.7 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.

3.5.8 Log viewer

Despite countless efforts (unit testing, test coverage…), DataLab might crash or behave unexpectedly.

For those situations, DataLab provides two logs (located in the DataLab configuration folder, e.g. `~/.DataLab_v1`):

- "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. 77: DataLab log viewer (see "?" menu)

3.5.9 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

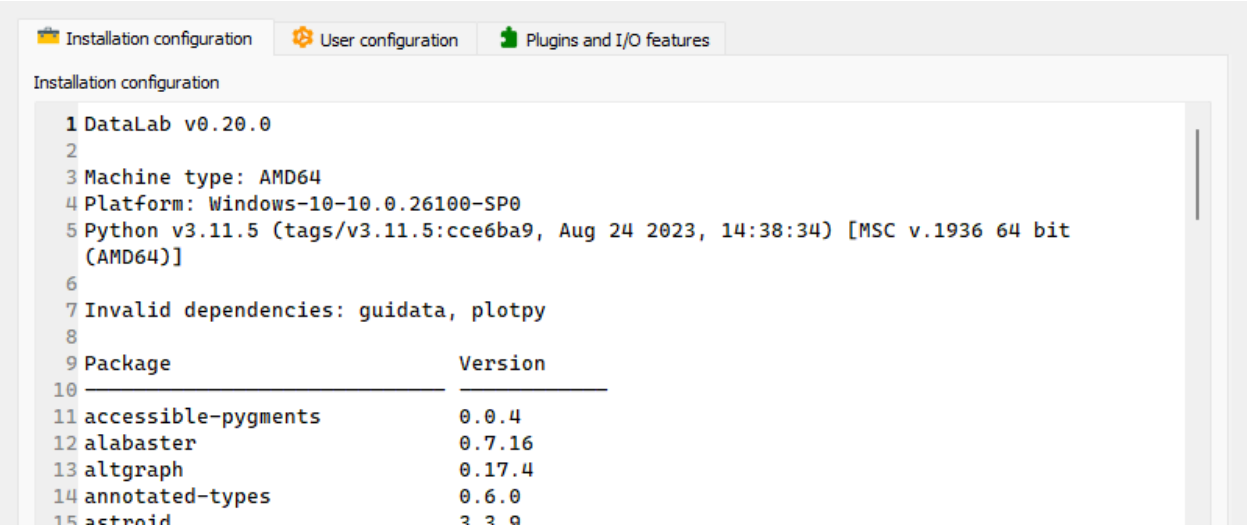


Fig. 78: Installation and configuration (see "?" menu), tab 1

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).

2. User Configuration

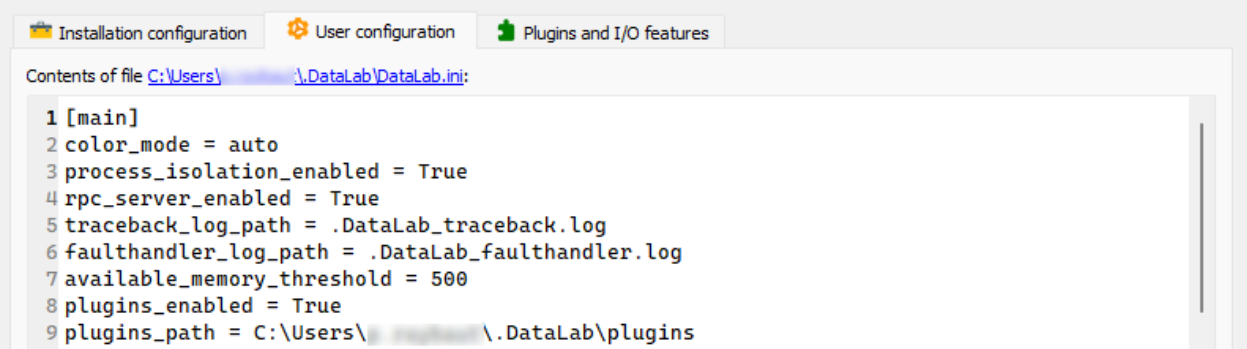


Fig. 79: Installation and configuration (see "?" menu), tab 2

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

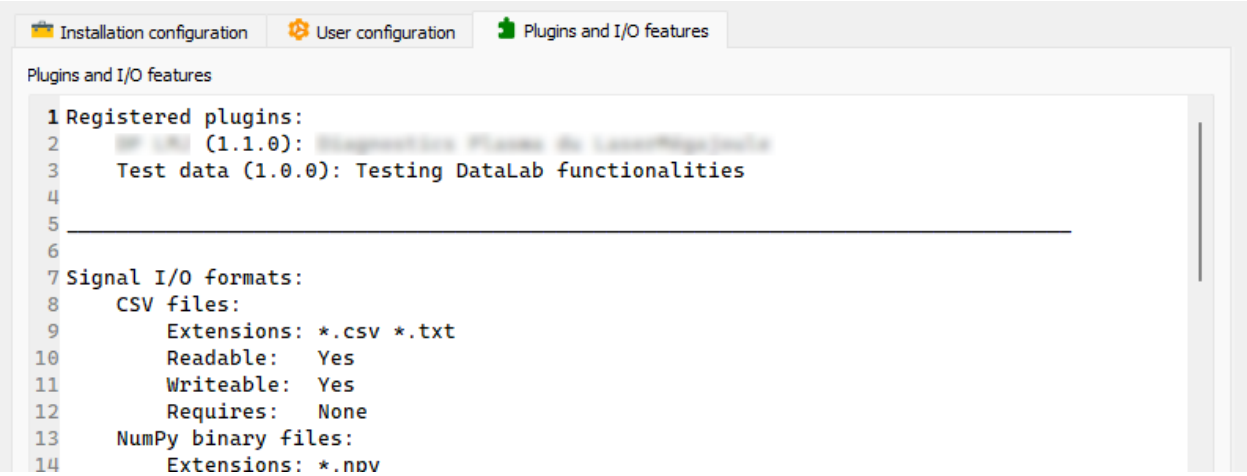


Fig. 80: Installation and configuration (see "?" menu), tab 3

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.

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).

3.5.10 Command line features

Run DataLab

To run DataLab from the command line, type the following:

```
$ datalab
```

To show help on command line usage, run the module from a terminal. On Windows, the `datalab` GUI entry point may not display console output:

```
$ python -m datalab.app --help
usage: app.py [-h] [-b path] [-v] [--reset] [--unattended] [--accept_dialogs]
              [--screenshot] [--screenshot_path SCREENSHOT_PATH]
              [--delay DELAY] [--xmlrpcport XMLRPCPORT]
              [--verbose {quiet,normal,debug}] [--smoke-test [REPORT]]
              [h5]

Run DataLab

positional arguments:
  h5                    HDF5 file names (separated by ';'), optionally with
                        dataset name (separated by ',')

options:
  -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
  --screenshot_path SCREENSHOT_PATH
                        path to save screenshots
  --delay DELAY         delay (ms) before quitting application in unattended
                        mode
  --xmlrpcport XMLRPCPORT
```

(continues on next page)

(continued from previous page)

```
                        XML-RPC port number
  --verbose {quiet,normal,debug}
                        verbosity level: for debugging/testing purpose
  --smoke-test [REPORT]
                        check the installation, write a JSON report (default:
                        datalab_smoke_test.json) and exit with a non-zero code
                        if a check failed
```

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
```

Check an installation

To check that DataLab is correctly installed, without opening its main window, use one of the following commands (`DataLab.exe` is the executable of the standalone version for Windows, in its installation folder):

```
$ datalab --smoke-test /path/to/report.json
$ DataLab.exe --smoke-test C:\path\to\report.json
```

DataLab checks the packages it relies on, the resources provided with it (macro templates, PDF documentation, French translations and icons) and processing in a separate process, writes the results in the JSON report and exits. Without a path, the report is written as `datalab_smoke_test.json` in the current folder, which must be writable. The exit code is 0 if every check passed, 1 otherwise: the report gives the cause of each failure.

> **Note**
>
> The standalone version shows no console output: read the report. If it cannot start at all, it exits with code 1 and writes the cause in `datalab_error.log`, in the current folder.

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 of the Sigima library, which provides the signal and image processing features of DataLab. The test functions including validation tests are marked with the `@pytest.mark.validation` decorator.

To execute DataLab technical validation tests, run the following:

```
$ pytest --pyargs sigima -m validation
```

> **See also**
>
> See section Validation for more information on DataLab's validation strategy.

Run complete test suite

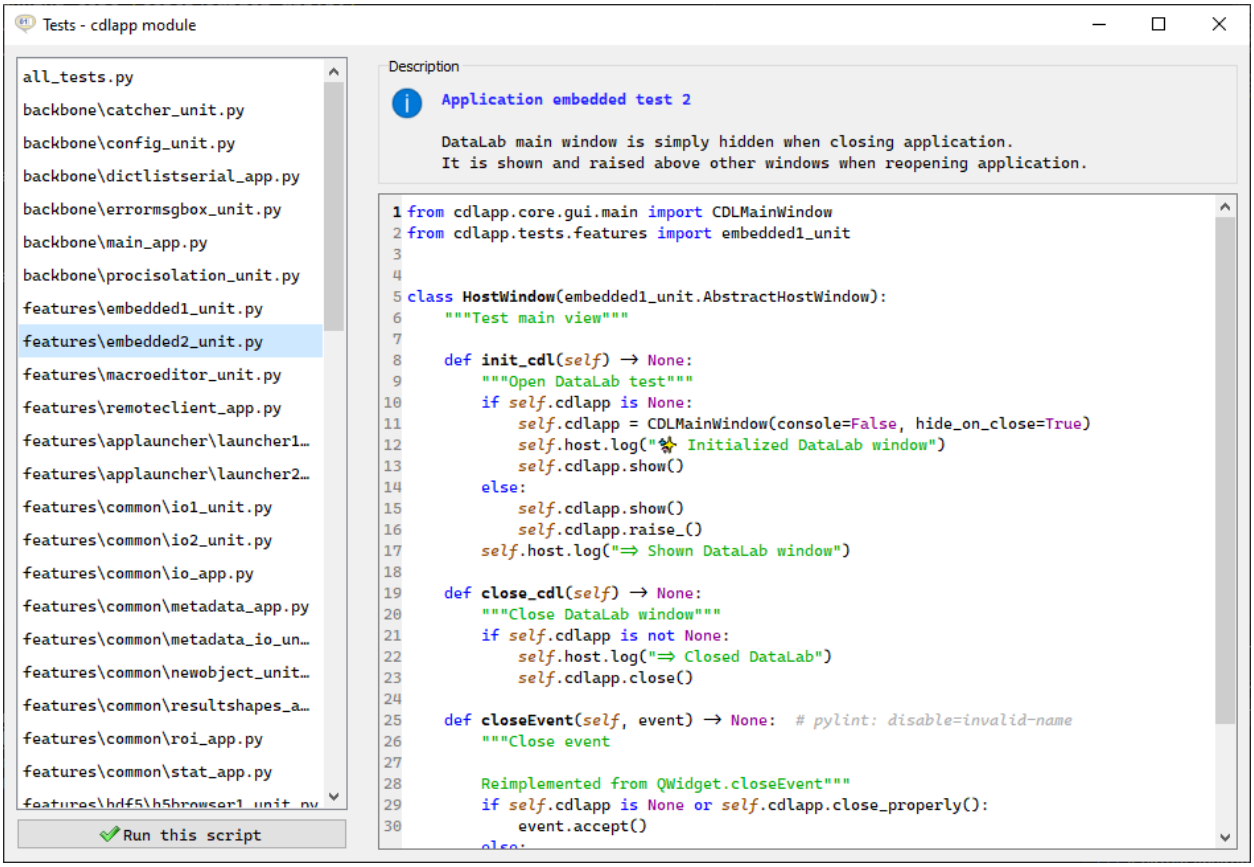
To execute all DataLab unit tests, simply run:

```
$ pytest
```

Run interactive tests

To execute DataLab interactive tests, run the following:

```
$ datalab-tests
```



3.5.11 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).

> **Note**
>
> A significant part of the DataLab graphical layer is provided by the sigimax package: generic main window, configuration system, dockable plot widgets, HDF5 workspace and browser, log viewer and scientific dialogs. Those classes are documented in the SigimaX documentation, and are the ones to look at when building a Qt scientific application on the same foundation (see The DataLab Platform).

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) → str

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.

Returns
UUID of the created group

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: 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

call_method(method_name: str, *args, panel: str | None = None, **kwargs)

Call a public method on a panel or main window.

Method resolution order when panel is None: 1. Try main window (DLMainWindow) 2. If not found, try current panel (BaseDataPanel)

Parameters

- **method_name** – Name of the method to call
- ***args** – Positional arguments to pass to the method
- **panel** – Panel name (“signal”, “image”, or None for auto-detection). Defaults to None.
- ****kwargs** – Keyword arguments to pass to the method

Returns
The return value of the called method

Raises

- **AttributeError** – If the method does not exist or is not public
- **ValueError** – If the panel name is invalid

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.

export_capsule(filename: str, name: str | None = None, description: str | None = None, as_revision: bool = True, workspace_ref: str | None = None, workspace_filename: str | None = None, attachments: list[str] | None = None) → str

Export the workspace as a capsule (workspace and its provenance).

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **name** – Capsule name.
- **description** – Capsule description.
- **as_revision** – When the workspace was opened from a capsule, name that capsule as the parent revision. Defaults to True.
- **workspace_ref** – Absolute IRI where the workspace will be published: the capsule then references the workspace instead of including it.
- **workspace_filename** – File name of the workspace published at workspace_ref (default: last segment of the IRI, next to the capsule).
- **attachments** – Light files declared in the capsule (16 MiB at most each), never opened or executed.

Returns
Manifest fingerprint of the capsule.

get_current_object_uuid() → str | None

Return current object uuid in current panel.

Returns
UUID of the current object, or None if no object is current.

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

get_webapi_status() → dict

Get Web API server status.

Returns
Dictionary with “running”, “url”, “token” and “available” keys.

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, param: DataSet | None = None) → None

Open objects from directory in current panel (signals/images).

Parameters

- **path** – directory path
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

load_from_files(filenames: list[str], param: DataSet | None = None) → None

Open objects from files in current panel (signals/images).

Parameters

- **filenames** – list of file names
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

load_h5_workspace(h5files: list[str], reset_all: bool = False) → None

Load native DataLab HDF5 workspace files without any GUI elements.

This method can be safely called from scripts (e.g., internal console, macros) as it does not create any Qt widgets, dialogs, or progress bars.

> **Warning**
>
> This method only supports native DataLab HDF5 files. For importing arbitrary HDF5 files (non-native), use *open_h5_files()* or *import_h5_file()* instead.

Parameters

- **h5files** – List of native DataLab HDF5 filenames
- **reset_all** – Reset all application data before importing. Defaults to False.

Raises

ValueError – If a file is not a valid native DataLab HDF5 file

open_capsule(filename: str, workspace: str | None = None) → None

Open a capsule as the workspace, replacing the current one.

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **workspace** – Workspace file of a capsule that references its workspace instead of including it.

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

remove_object(force: bool = False) → None

Remove current object from current panel.

Parameters

force – if True, remove object without confirmation. Defaults to False.

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_h5_workspace(filename: str) → None

Save current workspace to a native DataLab HDF5 file without GUI elements.

This method can be safely called from scripts (e.g., internal console, macros) as it does not create any Qt widgets, dialogs, or progress bars.

Parameters
filename – HDF5 filename to save to

Raises
IOError – If file cannot be saved

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_object(obj: SignalObj | ImageObj) → None

Set object data in DataLab.

Update an existing object in DataLab with new data from `obj`. The object is identified by its UUID (which is carried by `obj` from a previous *`get_object()`* call).

Parameters
obj – Signal or image object (must have the same UUID as an existing object in DataLab)

Raises
KeyError – if no object with matching UUID is found

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.

start_webapi_server(host: str | None = None, port: int | None = None) → dict

Start the Web API server.

Parameters

- **host** – Host address to bind to. Defaults to "127.0.0.1".
- **port** – Port number. Defaults to auto-detect available port.

Returns
Dictionary with "url" and "token" keys.

Raises
RuntimeError – If Web API dependencies not installed or server already running.

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

stop_webapi_server() → None

Stop the Web API server.

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 | None = None, input_source: Literal['local', 'plugin'] = 'local')

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.

- **input_source** – Source of objects added through this proxy.

add_signal(title: str, xdata: np.ndarray, ydata: np.ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True, new_session_behavior: SessionBehavior | None = None) → 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
- **new_session_behavior** – Optional history session creation policy

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(title: str, data: np.ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True, new_session_behavior: SessionBehavior | None = None) → 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
- **new_session_behavior** – Optional history session creation policy

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, new_session_behavior: SessionBehavior | None = None) → bool

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
- **new_session_behavior** – Optional history session creation policy

Returns
True if the object was added successfully, False otherwise

multiload_session(panel: Literal['signal', 'image'], new_session_behavior: SessionBehavior | None = None) → Generator[None, None, None]

Apply one lazy session decision to a multi-object load.

Parameters

- **panel** – Target data panel ("signal" or "image")
- **new_session_behavior** – Optional history session creation policy

Raises

- **ValueError** – If the panel or session behavior is invalid
- **RuntimeError** – If another multiload session is already active

calc(name: str, param: 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

load_h5_workspace(h5files: list[str] | str, reset_all: bool = False) → None

Load HDF5 workspace files without showing file dialog.

This method loads one or more DataLab native HDF5 files directly, bypassing the file dialog. It is safe to call from the internal console or any context where Qt dialogs would cause threading issues.

Parameters

- **h5files** – Path(s) to HDF5 file(s). Can be a single path string or a list of paths
- **reset_all** – If True, reset workspace before loading. Defaults to False.

Raises
ValueError – if file is not a DataLab native HDF5 file

save_h5_workspace(filename: str) → None

Save workspace to HDF5 file without showing file dialog.

This method saves the current workspace to a DataLab native HDF5 file directly, bypassing the file dialog. It is safe to call from the internal console or any context where Qt dialogs would cause threading issues.

Parameters
filename – Path to the output HDF5 file

add_group(title: str, panel: str | None = None, select: bool = False) → str

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.

Returns
UUID of the created group

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.

call_method(method_name: str, *args, panel: str | None = None, **kwargs)

Call a public method on a panel or main window.

Method resolution order when panel is None: 1. Try main window (DLMainWindow) 2. If not found, try current panel (BaseDataPanel)

Parameters

- **method_name** – Name of the method to call
- ***args** – Positional arguments to pass to the method
- **panel** – Panel name ("signal", "image", or None for auto-detection). Defaults to None.
- ****kwargs** – Keyword arguments to pass to the method

Returns
The return value of the called method

Raises

- **AttributeError** – If the method does not exist or is not public
- **ValueError** – If the panel name is invalid

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.

export_capsule(filename: str, name: str | None = None, description: str | None = None, as_revision: bool = True, workspace_ref: str | None = None, workspace_filename: str | None = None, attachments: list[str] | None = None) → str

Export the workspace as a capsule (workspace and its provenance).

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **name** – Capsule name.
- **description** – Capsule description.
- **as_revision** – When the workspace was opened from a capsule, name that capsule as the parent revision. Defaults to True.
- **workspace_ref** – Absolute IRI where the workspace will be published: the capsule then references the workspace instead of including it.
- **workspace_filename** – File name of the workspace published at workspace_ref (default: last segment of the IRI, next to the capsule).
- **attachments** – Light files declared in the capsule (16 MiB at most each), never opened or executed.

Returns
Manifest fingerprint of the capsule.

get_current_object_uuid() → str | None

Return current object uuid in current panel.

Returns
UUID of the current object, or None if no object is current.

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

get_webapi_status() → dict

Get Web API server status.

Returns
Dictionary with "running", "url", "token" and "available" keys.

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, param: DataSet | None = None) → None

Open objects from directory in current panel (signals/images).

Parameters

- **path** – directory path
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

load_from_files(filenames: list[str], param: DataSet | None = None) → None

Open objects from files in current panel (signals/images).

Parameters

- **filenames** – list of file names
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog (e.g. `sigima.io.RawImageImportParam` for RAW images). Defaults to None.

open_capsule(filename: str, workspace: str | None = None) → None

Open a capsule as the workspace, replacing the current one.

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **workspace** – Workspace file of a capsule that references its workspace instead of including it.

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

remove_object(force: bool = False) → None

Remove current object from current panel.

Parameters
force – if True, remove object without confirmation. Defaults to False.

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_object(obj: SignalObj | ImageObj) → None

Set object data in DataLab.

Update an existing object in DataLab with new data from `obj`. The object is identified by its UUID (which is carried by `obj` from a previous *`get_object()`* call).

Parameters
obj – Signal or image object (must have the same UUID as an existing object in DataLab)

Raises
KeyError – if no object with matching UUID is found

start_webapi_server(host: str | None = None, port: int | None = None) → dict

Start the Web API server.

Parameters

- **host** – Host address to bind to. Defaults to "127.0.0.1".
- **port** – Port number. Defaults to auto-detect available port.

Returns
Dictionary with "url" and "token" keys.

Raises
RuntimeError – If Web API dependencies not installed or server already running.

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

stop_webapi_server() → None
Stop the Web API server.

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.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 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
- **ValueError** – if multiple objects have the same title
- **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

resolve_title_reference(reference: str) → tuple[str, SignalObj | ImageObj | ObjectGroup] | None

Resolve a title reference to `(panel_str, object or group)`.

Parameters
reference – `<UUID8>` (object) or `g<UUID8>` (group) reference

Returns
The unique match across signal and image panels, or None if the reference matches nothing or is ambiguous.

render_object_title(title: str) → str

Render an object title according to user settings.

refresh_object_title_displays() → None

Refresh object titles in trees and shown plot items.

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.

get_current_object_uuid() → str | None

Return current object uuid in current panel.

Returns
UUID of the current object, or None if no object is current.

add_group(title: str, panel: Literal['signal', 'image'] | None = None, select: bool = False) → str

Add group to DataLab.

Parameters

- **title** – Group title
- **panel** – Panel name. Defaults to None.
- **select** – Select the group after creation. Defaults to False.

Returns
UUID of the created group

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.

call_method(method_name: str, *args, panel: Literal['signal', 'image'] | None = None, **kwargs)
Call a public method on a panel or main window.

This generic method allows calling any public method that is not explicitly exposed in the proxy API. The method resolution follows this order:

1. If panel is specified: call method on that specific panel
2. If panel is None: a. Try to call method on main window (DLMainWindow) b. If not found, try to call method on current panel (BaseDataPanel)

This makes it convenient to call panel methods without specifying the panel parameter when working on the current panel.

Parameters

- **method_name** – Name of the method to call
- ***args** – Positional arguments to pass to the method
- **panel** – Panel name (“signal”, “image”, or None for auto-detection). Defaults to None.
- ****kwargs** – Keyword arguments to pass to the method

Returns
The return value of the called method

Raises

- **AttributeError** – If the method does not exist or is not public
- **ValueError** – If the panel name is invalid

Examples

```
>>> # Call remove_object on current panel (auto-detected)
>>> win.call_method("remove_object", force=True)
>>> # Call a signal panel method specifically
>>> win.call_method("delete_all_objects", panel="signal")
>>> # Call main window method
>>> win.call_method("get_current_panel")
```

remote_request_context() → Iterator[None]

Context in which a request of a remote client runs (XML-RPC, Web API).

Errors are propagated to the caller, which reports them to the client, instead of being shown in a message box: no user waits for it, and showing a message box crashes DataLab when it runs without display. The error is also shown in the status bar.

call_method_slot(method_name: str, args: list, panel: Literal['signal', 'image'] | None, kwargs: dict) → None

Slot to call a method from RemoteServer thread in GUI thread.

This slot receives signals from RemoteServer and executes the method in the GUI thread, avoiding thread-safety issues with Qt widgets and dialogs.

Parameters

- **method_name** – Name of the method to call
- **args** – Positional arguments as a list
- **panel** – Panel name or None for auto-detection
- **kwargs** – Keyword arguments as a dict

run_remote_command(command: Callable[..., object], *args, return_result: bool = False) → None

Slot to run a command requested by a remote client in GUI thread.

The exception raised by the command, if any, is stored in RemoteServer to be re-raised to the client. Being remote controlled, this slot signals that DataLab is ready only once the command is done and its exception stored.

Parameters

- **command** – Main window method running the command
- **args** – Command arguments
- **return_result** – If True, store the command result in RemoteServer, to be returned to the client (it must be XML-RPC serializable)

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.

start_webapi_server(host: str | None = None, port: int | None = None) → dict

Start the Web API server.

Parameters

- **host** – Host address to bind to. Defaults to "127.0.0.1".
- **port** – Port number. Defaults to auto-detect available port.

Returns
Dictionary with "url" and "token" keys.

Raises
RuntimeError – If Web API imports are unavailable or the server is running.

stop_webapi_server() → None

Stop the Web API server.

get_webapi_status() → dict

Get Web API server status.

Returns
Dictionary with "running", "url", "token" and "available" keys.

property panels: tuple[*AbstractPanel*, ...]

Return the tuple of implemented panels (signal, image, macro, history)

Returns
Tuple of panels

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_menu_screenshots() → None

Take menu screenshots

open_plugin_instrument(plugin: PluginBase, tool: PluginTool) → InstrumentWindow

Show the instrument window of a plugin tool, creating it if needed.

The successive windows of an instrument share its worker: a window opened again calls the instrument once the closed one has closed it.

close_plugin_instruments() → None

Close all plugin instrument windows, and wait for their instruments, including those of windows closed before, within a common time limit.

show_plugin_catalog() → None

Open the plugin configuration dialog on its Catalog tab

show_applications(plugin_id: str | None = None) → None

Show the Applications panel

The first time, the panel floats beside the main window; afterwards, it is shown where the user left it (docked, tabified or floating). When it is already shown with the requested application, it flashes to draw attention to it; switching to another application needs no flashing.

Parameters
plugin_id – ID of the application plugin to show (default: the one already shown)

set_plugins_enabled(enabled: bool) → None

Apply the global third-party plugin enabled state.

reload_plugins() → None

Reload third-party plugins at runtime.

This unregisters active plugins, clears plugin actions from both panels, re-discovers plugin modules (reloading code changes from disk), re-registers enabled plugins, then recreates plugin actions and refreshes menus.

show_command_palette() → None

Show the command palette (searchable list of menu commands).

Lists every command available for the current panel by its menu path and triggers the one chosen by the user.

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

current_panel_name() → str

Return current panel name, without processing pending events

Unlike *get_current_panel()*, this may be called while Qt is handling an event, e.g. from a show event (see issue #527).

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: DataSet | None = None, edit: bool = True) → 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.
- **edit** – Whether to show parameter edit dialog. Defaults to True. Set to False when calling from remote/API to avoid blocking dialogs.

Raises
FeatureNotFoundError – unknown function (a `ValueError`)

has_objects() → bool

Return True if sig/ima panels have any object

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

verify_provenance_activity(activity_id: str) → dict

Recompute a recorded activity as a separate candidate and compare it.

Nothing is inserted in the workspace and the ledger is not modified.

Parameters
activity_id – Identifier of the recorded activity.

Returns
Verification report (JSON-compatible).

show_capsule_dialog() → None

Show the workspace capsule dialog (export, inspect, replay, compare).

export_capsule(filename: str, name: str | None = None, description: str | None = None, as_revision: bool = True, workspace_ref: str | None = None, workspace_filename: str | None = None, attachments: list[str] | None = None) → str

Export the workspace as a capsule (workspace and its provenance).

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **name** – Capsule name.
- **description** – Capsule description.
- **as_revision** – When the workspace was opened from a capsule, name that capsule as the parent revision. Defaults to True.
- **workspace_ref** – Absolute IRI where the workspace will be published: the capsule then references the workspace instead of including it, and the workspace is saved apart, to workspace_filename.
- **workspace_filename** – File name of the workspace published at workspace_ref (default: last segment of the IRI, next to the capsule).

- **attachments** – Light files (protocol, notes, small tables, 16 MiB at most each) declared in the capsule, never opened or executed.

Returns
Manifest fingerprint of the capsule.

open_capsule(filename: str, workspace: str | None = None) → None

Open a capsule as the workspace, replacing the current one.

The capsule is checked first (archive, manifest, workspace SHA-256 and size): the workspace is left unchanged if it is invalid. A revision exported later names this capsule as its parent.

Parameters

- **filename** – Capsule file name (`.dlcapsule`).
- **workspace** – Workspace file of a capsule that references its workspace instead of including it.

remove_object(force: bool = False) → None

Remove current object from current panel.

Parameters
force – if True, remove object without confirmation. Defaults to False.

save_to_h5_file(filename: str | None = 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.

When the question about clearing the workspace will be asked, the files are chosen first: if a workspace file holds provenance, the question says that appending it ignores that provenance (no further warning).

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

import_dataset_from_file(filename: str, dsetname: str | None, import_all: bool | None, reset_all: bool) → None

Open a DataLab workspace file, or import a single dataset from it

Parameters

- **filename** – Path to the HDF5 file (already validated)
- **dsetname** – Dataset name to import, or `None` to import all
- **import_all** – If `True`, import all datasets without browsing
- **reset_all** – If `True`, clear workspace before importing

import_all_from_h5_file(filename: str, reset_all: bool | None = None) → None

Import an HDF5 file through the DataLab HDF5 browser

Parameters

- **filename** – HDF5 filename
- **reset_all** – Delete all DataLab signals/images before importing data

browse_h5_files(filenames: list[str], reset_all: bool | None = None) → None

Browse HDF5 files

Parameters

- **filenames** – HDF5 filenames
- **reset_all** – Reset all application data before importing

load_h5_workspace(h5files: list[str], reset_all: bool = False) → None

Load native DataLab HDF5 workspace files programmatically.

This method does not create file-selection widgets or progress bars. When history recording is active and the new-session policy is `"ask"`, a history-session question may still be shown before loading.

When a file holding provenance is appended (`reset_all=False`), its provenance is not imported: no question is asked, and the notice `datalab_capsule.recorder.APPEND_NOTICE` is added to `provenance.notices` (and logged).

> **Warning**
>
> This method only supports native DataLab HDF5 files. For importing arbitrary HDF5 files (non-native), use the GUI menu or macros with *datalab.control.proxy.RemoteProxy*.

Parameters

- **h5files** – List of native DataLab HDF5 filenames
- **reset_all** – Reset all application data before importing. Defaults to False.

Raises

ValueError – If a file is not a valid native DataLab HDF5 file

save_h5_workspace(filename: str) → None

Save current workspace to a native DataLab HDF5 file without GUI elements.

This method can be safely called from the internal console as it does not create any Qt widgets, dialogs, or progress bars. It is designed for programmatic use when saving DataLab workspace files.

Parameters

filename – HDF5 filename to save to

Raises

IOError – If file cannot be saved

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, new_session_behavior: SessionBehavior | None = None) → bool

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
- **new_session_behavior** – Optional history session creation policy

Returns
True if the object was added successfully, False otherwise

set_object(obj: SignalObj | ImageObj) → None

Set object data - update an existing signal or image in-place.

The existing object is identified by UUID carried by `obj` (from a previous *`get_object()`* call).

Parameters
obj – object with updated data (signal or image)

Raises

- **KeyError** – if no object with matching UUID is found
- **TypeError** – if object type is unsupported

load_from_files(filenames: list[str], param: DataSet | None = None) → None

Open objects from files in current panel (signals/images)

Parameters

- **filenames** – list of filenames
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog. Defaults to None.

load_from_directory(path: str, param: DataSet | None = None) → None

Open objects from directory in current panel (signals/images).

Parameters

- **path** – directory path
- **param** – Optional format-specific import parameters applied to every file requiring them, without showing any dialog. Defaults to None.

get_version() → str

Return DataLab public version.

Returns
DataLab version

add_signal(title: str, xdata: np.ndarray, ydata: np.ndarray, xunit: str = '', yunit: str = '', xlabel: str = '', ylabel: str = '', group_id: str = '', set_current: bool = True, new_session_behavior: SessionBehavior | None = None) → 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
- **new_session_behavior** – Optional history session creation policy

Returns
True if signal was added successfully, False otherwise

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(title: str, data: np.ndarray, xunit: str = '', yunit: str = '', zunit: str = '', xlabel: str = '', ylabel: str = '', zlabel: str = '', group_id: str = '', set_current: bool = True, *, new_session_behavior: SessionBehavior | None = None) → 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
- **new_session_behavior** – Optional history session creation policy

Returns
True if image was added successfully, False otherwise

Raises
ValueError – Invalid data dtype

show_settings(tab: str | None = None) → None

Show the settings dialog, then apply the changed settings

Parameters

tab – Key of the settings tab to show first (e.g. "ai"), or None for the first tab

play_demo() → None

Play demo

show_tour() → None

Show tour

show_welcome_page() → None

Show the welcome page and bring it to the front

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.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.**collect_metadata_keys**(objs: Sequence[TypeObj]) → list[tuple[str, str]]

Return the user-visible scalar metadata keys of objects.

Parameters

objs – objects to inspect

Returns

Sorted `(key, description)` pairs, the description showing an example value; internal options, ROIs and analysis results are excluded.

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.**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 the provenance of an object in the Provenance tab.

When the workspace ledger records a production, an old record or an analysis of the object, the provenance view is shown (see datalab.widgets.provenance.ProvenanceWidget); otherwise, the history rebuilt from the processing metadata of the object is shown.

Parameters
obj – Signal or Image object

Returns
True if a history was found and displayed, False otherwise.

refresh_history() → None

Show again the History of the current object (ledger read again).

Called when activities were recorded or the workspace changed: results of data checks and verifications shown before are dropped.

update_properties_from(obj: SignalObj | ImageObj | None = None, force_tab: Literal['creation', 'processing', 'analysis', None] | None = None) → None

Update properties panel (properties, creation, processing) from object.

Parameters

- **obj** – Signal or Image object
- **force_tab** – Force a specific tab to be current

keeping_tab(kind: str) → Generator[None, None, None]

Keep the tab of kind current when another object is shown meanwhile.

For navigation from a tab to another object (e.g. from the provenance of a result to one of its inputs), which otherwise shows Properties.

mark_as_newly_created(obj: SignalObj | ImageObj) → None

Mark object to show Creation tab on next selection.

Parameters
obj – Object to mark

mark_as_freshly_processed(obj: SignalObj | ImageObj) → None

Mark object to show Processing tab on next selection.

Parameters
obj – Object to mark

mark_as_fresh_analysis(obj: SignalObj | ImageObj) → None

Mark object to show Analysis tab on next selection.

Parameters
obj – Object to mark

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

setup_legacy_creation_tab(obj: SignalObj | ImageObj, error: str, set_current: bool = False) → bool

Setup actions for explicitly converting historical peak parameters.

setup_invalid_creation_tab(obj: SignalObj | ImageObj, error: str, set_current: bool = False) → bool

Show a read-only state for invalid creation metadata.

convert_current_creation_parameters() → None

Convert the current object's historical peak creation parameters.

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 to show and apply again a processing.

The call that produced the object is restored (see `datalab.processingcall.resolve_processing_call()`): 1-to-1, 2-to-1 and n-to-1 processing, with every input. Parameters are edited in a parameter editor; a processing without editable parameters gets an explicit Reapply button. When the processing cannot be applied again (deleted input, unavailable operation…), the reason is shown.

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

reapply_processing(obj: SignalObj | ImageObj | None = None) → None

Apply again the processing of an object without editable parameters.

Parameters
obj – Processed object (default: the object of the Processing tab).

setup_analysis_tab(obj: SignalObj | ImageObj, set_current: bool = False) → bool

Setup the Analysis tab with parameter editor for re-running analysis.

This tab lets the user edit the parameters of a 1-to-0 analysis operation (peak detection, FWHM, segments, etc.) and re-run it in place with the modified parameters.

Parameters

- **obj** – Signal or Image object
- **set_current** – If True, set the Analysis tab as current after creation

Returns
True if Analysis tab was set up, False otherwise

apply_analysis_parameters(obj: SignalObj | ImageObj | None = None, interactive: bool = True, param: DataSet | None = None) → bool

Apply analysis parameters: re-run the 1-to-0 analysis in place.

Parameters

- **obj** – Signal or Image object to re-analyze. If None, uses the current analysis object.
- **interactive** – If True, show error messages in the UI.
- **param** – Explicit analysis parameters to apply. When None (default), fall back to the editor dataset or the stored analysis parameters.

apply_processing_parameters(obj: SignalObj | ImageObj | None = None, interactive: bool = True, param: DataSet | None = None) → ProcessingReport

Apply processing parameters by recomputing the existing object in place.

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.

- **param** – Explicit processing parameters to apply. When provided, this takes precedence and makes the call independent of the Processing tab editor state (used e.g. by programmatic recompute paths). When None (default), fall back to the editor dataset or the stored processing parameters.

Returns
ProcessingReport with success status, object UUID, and optional message.

apply_recomputed_object_in_place(obj: SignalObj | ImageObj, new_obj: SignalObj | ImageObj, proc_params: ProcessingParameters) → None

Apply a freshly recomputed object onto `obj` in place.

Copies scientific data, coordinates, labels and units while preserving metadata, annotations and display settings. Only processing metadata changes.

Parameters

- **obj** – Existing object to update in place (identity preserved).
- **new_obj** – Freshly recomputed object providing title + data.
- **proc_params** – Updated processing parameters to store on `obj`.

class datalab.gui.panel.base.**H5ImportBatch**(panel: BaseDataPanel, reference_remap: dict[str, str], objects: list[TypeObj], groups: list[ObjectGroup])

Objects and UUID remapping created by one native HDF5 import.

apply_reference_remap(reference_remap: dict[str, str], title_remap: dict[str, str]) → None

Remap title and processing references in this imported batch.

Parameters

- **reference_remap** – serialized UUID to imported UUID mapping
- **title_remap** – serialized title reference to imported title reference

classmethod finalize_imports(batches: list[H5ImportBatch], refresh_panels: bool = False) → None

Apply the combined signal/image UUID remapping to imported batches.

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 only UUIDs that conflict with existing objects (object import).

abstractmethod serialize_to_hdf5(writer: NativeH5Writer) → None

Serialize whole panel to a HDF5 file

abstractmethod deserialize_from_hdf5(reader: NativeH5Reader, reset_all: bool = False) → H5ImportBatch | None

Deserialize whole panel from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate only UUIDs that conflict with existing objects (object import).

abstractmethod create_object() → ObjItf

Create and return object

abstractmethod add_object(obj: ObjItf) → None

Add object to panel

abstractmethod remove_all_objects()

Remove all objects

class datalab.gui.panel.base.**PasteMetadataParam**(title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False, skip_defaults: bool = False)

Paste metadata parameters

classmethod for_dialog() → PasteMetadataParam

Create parameters with the metadata paste dialog's explanation.

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**(objs: list[TypeObj] | None = None, extensions: list[str] | None = None, labels: list[str] | None = None)

Save to directory parameters

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**(objs: list[TypeObj] | None = None, known_keys: Sequence[tuple[str, str]] | None = None)

Add metadata parameters

Parameters

- **objs** – objects receiving the metadata item
- **known_keys** – suggested `(key, description)` pairs

on_help_button_click(_item: ButtonItem, _value: None, parent: QWidget) → None

Help button callback.

get_known_key_choices(_item=None, _value=None)

Return the suggested metadata keys.

on_known_key_changed(_item=None, value=None) → None

Copy the selected suggestion into the metadata key.

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 | None]

Build values according to current parameters.

Returns

One value per object; `None` for objects left unchanged because the extraction pattern does not match.

Raises

ValueError – If the extraction pattern is invalid or does not match (when asked to report it), or if a value cannot be converted.

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

abstractmethod static get_roi_class() → Type[TypeROI]

Return ROI class

abstractmethod 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) → H5ImportBatch

Deserialize whole panel from a HDF5 file

Parameters

- **reader** – HDF5 reader
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate only UUIDs that conflict with existing objects (object import).

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

add_objects(objects: Sequence[TypeObj], group_id: str | None = None, set_current: bool = True) → None

Add multiple objects atomically.

Parameters

- **objects** – Objects to add, in insertion order
- **group_id** – Group ID to which the objects belong. If None or empty, the objects are added to the current group.
- **set_current** – If True, set the last added object as current

set_object(obj: TypeObj) → None

Update an existing object in-place with data from `obj`.

The existing object is identified by UUID (carried by `obj` from a previous `get_object()` call). All data attributes are copied from `obj` to the existing object while preserving internal metadata (number, group membership).

Parameters
obj – SignalObj or ImageObj with the same UUID as an existing object.

Raises
KeyError – if no object with matching UUID is found

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, group_uuid: str | None = None) → ObjectGroup

Add group

Parameters

- **title** – group title
- **select** – if True, select the group in the tree view. Defaults to False.
- **group_uuid** – optional group UUID. If None, a new UUID is generated.

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)

get_known_metadata_keys(objs: Sequence[TypeObj]) → list[tuple[str, str]]

Return the metadata keys suggested by the Add metadata dialog.

Parameters
objs – selected objects

Returns
the keys found on the objects, then the keys expected by the methods of application plugins

Return type
`(key, description)` pairs

add_metadata(param: AddMetadataParam | None = None) → None

Add metadata item to selected object(s)

Parameters
param – Add metadata parameters

copy_roi(roi_data=None) → None

Copy regions of interest

Parameters
roi_data – ROI snapshot kept for legacy session replay compatibility. When `None` (interactive use), the ROI is read from the currently selected object.

paste_roi(roi_data=None) → None

Paste regions of interest

Parameters
roi_data – ROI snapshot kept for legacy session replay compatibility. When `None` (interactive use), the clipboard populated by *`copy_roi()`* is used.

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)

abstractmethod 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

abstractmethod 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

edit_import_parameters(filenames: list[str], param: DataSet | None = None) → list[tuple[str, DataSet | None]] | None

Return files to import with their optional format-specific parameters.

Parameters

- **filenames** – File names to import.
- **param** – Optional import parameters applied to every file requiring parameters, without showing any dialog.

Returns

File names with aligned import parameters, without the files whose parameter dialog was cancelled, or None if the whole import was cancelled.

Raises

TypeError – If `param` is given, as this panel has no import parameters.

apply_import_parameters(objs: list[SignalObj] | list[ImageObj], param: DataSet | None) → None

Apply optional import parameters to newly read objects.

Parameters

- **objs** – Imported objects.
- **param** – Optional format-specific import parameters.

edit_export_parameters(obj: TypeObj, filename: str) → tuple[bool, ImageExportParam | None]

Edit optional object export parameters.

Parameters

- **obj** – Object to export
- **filename** – File name

Returns

A tuple containing the confirmation state and optional export parameters

write_object_to_file(obj: TypeObj, filename: str, param: ImageExportParam | None = None) → None

Save object to file (signal/image).

Parameters

- **obj** – object
- **filename** – file name
- **param** – optional image export parameters

load_from_directory(directory: str | None = None, import_filenames: list[str] | None = None, import_params: list[DataSet] | None = None, param: DataSet | None = None, skipped_filenames: list[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
- **import_filenames** – Optional file names, relative to `directory`, of the files imported with format-specific parameters. Recorded by history so that replay does not reopen import dialogs.
- **import_params** – Optional format-specific parameters aligned with `import_filenames`.
- **param** – Optional format-specific parameters applied to every file requiring import parameters (e.g. RAW images), without showing any dialog.
- **skipped_filenames** – Optional file names, relative to `directory`, of the files whose import dialog was cancelled. Recorded by history so that replay skips them without reopening their dialog.

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, import_params: list[DataSet | None] | None = None, param: DataSet | None = None) → list[TypeObj]

Open and return objects from files, optionally adding them to DataLab.

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.
- **import_params** – Optional format-specific parameters aligned with file names. When provided by history replay, import dialogs are not reopened.
- **param** – Optional format-specific parameters applied to every file requiring import parameters (e.g. RAW images), without showing any dialog. Cannot be combined with `import_params`.

Returns
list of new objects

save_to_files(filenames: list[str] | str | None = None, export_params: list[ImageExportParam | None] | None = None, object_uuids: list[str] | None = None) → None

Save selected objects to files (signal/image).

Parameters

- **filenames** – File names
- **export_params** – Optional image export parameters aligned with file names
- **object_uuids** – Optional object UUIDs aligned with 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

refresh_after_reference_remap() → None

Refresh cached presentation after imported references are remapped.

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_selected() → None

Recompute/rerun selected objects or group with stored parameters.

This method handles both single objects and groups. For each object, it recomputes, on demand:

- the processing that produced it (1-to-1, 2-to-1 or n-to-1, as restored by `datalab.processingcall.resolve_processing_call()`: from the provenance ledger, else from the processing metadata), in place, and
- 1-to-0 analysis operations (statistics, measurements, detections, etc.).

Analysis results are not recomputed automatically when data, ROIs or object properties change: this manual action is the single, explicit entry point to refresh them. Objects without recomputable parameters are skipped.

recompute_1_to_1_objects(objects: list[SignalObj | ImageObj]) → tuple[set[str], bool]

Recompute in-place 1-to-1 processing operations for the given objects.

Parameters
objects – Objects with stored 1-to-1 processing parameters

Returns

- Set of object UUIDs successfully recomputed
- True if operation was interrupted (progress canceled or user chose to stop after a failure), False otherwise

Return type
Tuple containing

recompute_1_to_0_objects(objects: list[SignalObj | ImageObj]) → tuple[set[str], bool]

Recompute 1-to-0 analysis operations for the given objects.

Parameters
objects – Objects with stored 1-to-0 analysis parameters

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: 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

datalab.gui.panel.signal.**LEGACY_AUTOMATIC_LINE_WIDTH = 1.0**

Line width stored by DataLab 1.3 or earlier for curves drawn with the default line width, or with a line width clamped for curves with many points

class datalab.gui.panel.signal.**SignalAnnotatedSegmentTool**(manager, setup_shape_cb: Callable | None = None, handle_final_shape_cb: Callable | None = None, shape_style: tuple[str, str] | None=None, toolbar_id: str = <class 'plotpy.tools.base.DefaultToolbarID'>, title: str | None = None, icon: str | None = None, tip: str | None = None, switch_to_default_tool: bool | None = None)

Annotated segment tool for signals

The length and the angle of a segment combine its X and Y extents, whose units generally differ on a signal: unlike on images, they are not displayed.

create_shape() → tuple[AnnotatedSegment, int, int]

Create an annotated segment without computations

Returns

Annotated segment and the indices of its handles

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

alias 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

deserialize_object_from_hdf5(reader: NativeH5Reader, name: str, reset_all: bool = False) → SignalObj

Deserialize object from a HDF5 file

DataLab 1.3 and earlier stored automatic curve styles in signal metadata: when loading a file saved by these versions, the colors and line styles of their automatic style cycle are removed, so that curves are given the current automatic styles. Other stored styles are kept. These versions also stored the line width of every curve: a line width of 1.0 (the default width, also used for curves with many points) is removed, so that curves are given the configured line width; other widths were chosen by the user and are kept.

Parameters

- **reader** – HDF5 reader
- **name** – Object name in HDF5 file
- **reset_all** – If True, preserve original UUIDs (workspace reload). If False, regenerate only UUIDs that conflict with existing objects (object import).

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 the curve styles of the selected signals

Colors and line styles chosen by the user are removed from the metadata of the selected signals, whose curves are given automatic styles again.

Image panel

datalab.gui.panel.image.**get_image_export_dtype_choices**(param: ImageExportGUIParam, _item: ChoiceItem, _value: str) → list[tuple[str, str, None]]

Return target dtype choices supported by the selected image format.

class datalab.gui.panel.image.**ImageExportGUIParam**(title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False, skip_defaults: bool = False)

Image export parameters restricted to the selected file format.

is_item_active(name: str) → bool

Return True if the item named `name` is taken into account.

get_format_options() → dict[str, object]

Return writer options exposed by this format parameter class.

Inactive options are replaced by their default value, as they are not taken into account for the current values of the other options.

to_export_param() → ImageExportParam

Return backend export parameters from the current GUI values.

Inactive parameters (e.g. gamma without normalization) are reset to their default value, as they are not taken into account.

Returns
Backend image export parameters

Raises
ValueError – If a format option value is invalid

datalab.gui.panel.image.**get_image_export_choice_label**(value: object) → str

Return a translated human-readable label for an option choice.

datalab.gui.panel.image.**serialize_image_export_sequence**(value: object, spec: ImageExportOptionSpec) → tuple[int, int] | tuple[float, float] | list[float] | None

Serialize a pair or list option from its editable GUI representation.

datalab.gui.panel.image.**create_image_export_option_item**(spec: ImageExportOptionSpec, extension: str = '') → DataItem

Create a guidata item for an image export capability option.

Parameters

- **spec** – Option specification
- **extension** – Canonical format extension, used for format-specific help

Returns
Data item editing the option

datalab.gui.panel.image.**get_image_export_param_class**(filename: str) → type[ImageExportGUIParam]

Return the cached GUI parameter class for an export format.

datalab.gui.panel.image.**is_raw_image_file**(filename: str) → bool

Return True if the file is a headerless RAW image requiring parameters.

Parameters
filename – File name.

Returns
True if the file has a `.raw` extension.

datalab.gui.panel.image.**create_image_export_gui_param**(obj: ImageObj, filename: str) → ImageExportGUIParam

Create format-aware image export parameters with practical defaults.

Parameters

- **obj** – Source image
- **filename** – Destination file name

Returns
Image export GUI parameters

datalab.gui.panel.image.**compute_image_export_preview**(source: ImageObj, param: ImageExportParam, filename: str) → ImageObj

Return the image as it will be read back from the exported file.

This function runs in the preview process, so its arguments must be picklable.

Parameters

- **source** – Image to export
- **param** – Image export parameters
- **filename** – Destination file name

Returns
Copy of the source image holding the exported data, displayed in gray levels

datalab.gui.panel.image.**format_image_export_error**(message: str) → str

Return only the error message of an expected export validation error.

Validation errors raised by Sigima (`ValueError`/`TypeError`) explain the invalid configuration: the traceback is only kept for unexpected errors.

Parameters
message – Error message or formatted traceback

Returns
Message to display

class datalab.gui.panel.image.**ImageExportDialog**(parent: QWidget | None, obj: ImageObj, filename: str, param: ImageExportGUIParam, controller_factory: Callable = <class 'datalab.gui.processor.preview.PreviewController'>)

Format-aware image export dialog with a live preview of the exported data.

static to_export_param(guiparam: ImageExportGUIParam) → ImageExportParam

Return backend export parameters from GUI parameters.

Parameters
guiparam – Format-specific GUI parameters

Returns
Backend image export parameters

Raises
ValueError – If a format option value is invalid

get_export_param() → ImageExportParam

Return backend parameters from the current GUI values.

accept() → None

Validate the export configuration before accepting.

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

alias 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

edit_import_parameters(filenames: list[str], param: DataSet | None = None) → list[tuple[str, DataSet | None]] | None

Return files to import with their RAW image import parameters.

When several RAW files are imported, the user first chooses whether the same parameters apply to all of them (a single dialog) or not (one dialog per file).

Parameters

- **filenames** – File names to import.
- **param** – Optional RAW image import parameters applied to every RAW file without showing any dialog (e.g. remote control).

Returns
File names with aligned import parameters, without the RAW files whose dialog was cancelled, or None if the whole import was cancelled.

Raises
TypeError – If `param` is not a RAW image import parameter set.

edit_raw_import_parameters(raw_files: list[str]) → dict[str, RawImageImportParam] | None

Edit RAW image import parameters, once for all files or per file.

Parameters
raw_files – RAW file names.

Returns
Parameters of the confirmed files, or None if the import was cancelled.

apply_import_parameters(objs: list[ImageObj], param: DataSet | None) → None

Apply RAW display semantics without changing imported values.

Parameters

- **objs** – Imported image objects.
- **param** – Optional format-specific import parameters.

edit_export_parameters(obj: ImageObj, filename: str) → tuple[bool, ImageExportParam | None]

Edit image export parameters for the selected file format.

Parameters

- **obj** – Image to export
- **filename** – Destination file name

Returns
A tuple containing the confirmation state and export parameters

write_object_to_file(obj: ImageObj, filename: str, param: ImageExportParam | None = None) → None

Write an image with optional format-aware export parameters.

Parameters

- **obj** – Image to export
- **filename** – Destination file name
- **param** – Optional image export parameters

register_contrast_editor(obj: ImageObj, editor: ImageROIEditor) → None

Register an image ROI editor for contrast synchronization.

apply_shared_contrast(obj: ImageObj, zmin: float, zmax: float, source: ImageROIEditor | None = None) → None

Apply a contrast change coming from another view.

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

closeEvent(event) → None

Persist splitter state on close.

showEvent(event) → None

Trigger the macro-recovery prompt on first show.

Deferred to the first showEvent (and re-scheduled via singleShot) so the splash screen has been finished before the QMessageBox appears — otherwise the dialog can pop up behind the splash.

add_macro() → Macro

Add macro, optionally with name

Returns
Macro object

add_macro_with_code(title: str, code: str, source: str | None = None) → Macro

Add a macro with a predefined title and code, without prompting.

This helper is used by the AI assistant to programmatically inject a generated macro into the panel.

Parameters

- **title** – Macro title (displayed in the tab).
- **code** – Python source code of the macro.
- **source** – Optional origin marker (e.g. `"import"`, `"ai"`, `"template:simple_macro"`) recorded in the recent cache.

Returns
The newly created macro.

add_blank_macro() → Macro

Add a new empty macro and return it.

add_macro_from_template(template: MacroTemplate) → Macro

Add a new macro initialised from template.

Parameters
template – Macro template description.

Returns
The newly created macro.

duplicate_macro(number_or_title: int | str | None = None) → Macro | None

Duplicate the given (or current) macro.

Parameters
number_or_title – Number of the macro (starting at 1) or title. Defaults to None (current macro).

Returns
The duplicated macro, or None if no source macro was found.

show_recent_dialog() → None

Open a dialog listing recent macros and allow reopening one.

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)

update_untitled_counter() → None

Update the untitled counter based on existing macro titles

This scans all macro titles to find the highest "macro_XX" number and updates the global counter to prevent duplicate names.

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) → 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: 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: 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.

clear_plugin_actions() → None

Clear plugin actions and submenus

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: 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) → 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) → 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: 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: 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.

clear_plugin_actions() → None

Clear plugin actions and submenus

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: 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) → 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 model

The *datalab.objectmodel* module defines the object data model used by the GUI to store signals, images and groups.

The model is based on a hierarchical tree of objects, with two levels:

- The top level is a list of groups (ObjectGroup instances)
- The second level is a list of objects (SignalObj or ImageObj instances)

The model is implemented by the ObjectModel class.

Object group

The ObjectGroup class represents a group of objects. It is a container for SignalObj and ImageObj instances.

class datalab.objectmodel.**ObjectGroup**(title: str, model: ObjectModel, prefix: str, group_uuid: str | None = None)

Represents a DataLab object group

Parameters

- **title** – group title
- **model** – object model
- **prefix** – prefix for short ID ("gs" for signal groups, "gi" for image groups)
- **group_uuid** – optional group UUID. If None, a new UUID is generated.

property title: str

Return group title

append(obj: SignalObj | ImageObj) → None

Append object to group

insert(index: int, obj: SignalObj | ImageObj) → None

Insert object at index

remove(obj: SignalObj | ImageObj) → None

Remove object from group

remove_objects(uuids: Collection[str]) → None

Remove objects from group, renumbering the objects once

Parameters

uuids – UUIDs of the objects to remove (other UUIDs are ignored)

clear() → None

Clear group

get_objects() → list[SignalObj | ImageObj]

Return objects in group

get_object_ids() → list[str]

Return object ids in group

Object model

The ObjectModel class is a container for ObjectGroup instances, as well as a container for SignalObj and ImageObj instances.

class datalab.objectmodel.**ObjectModel**(group_prefix: str)

Represents a DataLab object model (groups of signals/images)

reset_short_ids() → None

Reset short IDs (used for object numbering)

This method is called when an object was removed from a group.

has_uuid(uuid: str) → bool

Check if an object or group with the given UUID exists in the model.

Parameters

uuid – UUID string to check

Returns

True if an object or group with this UUID exists, False otherwise

clear() → None

Clear model

get_all_objects(flatten: bool = True) → list[SignalObj | ImageObj] | list[list[SignalObj | ImageObj]]

Return all objects, in order of appearance in groups

Parameters

flatten – if True, return a flat list of objects, otherwise return a list of lists (one list per group)

Returns

List of objects in order of appearance in groups.

get_object_or_group(uuid: str) → SignalObj | ImageObj | ObjectGroup

Return object or group with uuid

find_by_title_reference(reference: str) → list[SignalObj | ImageObj | ObjectGroup]

Return the objects or groups matching a title reference.

Parameters

reference – <UUID8> (object) or g<UUID8> (group) reference

Returns

Matching objects or groups (more than one only on prefix collision).

get_group(uuid: str) → ObjectGroup

Return group with uuid

get_number(obj_or_group: SignalObj | ImageObj | ObjectGroup) → int

Return number of object or group

get_group_from_number(number: int) → ObjectGroup

Return group from its number.

Parameters
number – group number (starts with 1)

Returns
Group

Raises
IndexError – if group with number not found

get_group_from_title(title: str) → ObjectGroup

Return group from its title.

Parameters
title – group title

Returns
Group

Raises

- **KeyError** – if group with title not found
- **ValueError** – if multiple groups have the same title

get_group_from_object(obj: SignalObj | ImageObj) → ObjectGroup

Return group containing object

Parameters
obj – object to find group for

Returns
Group

Raises
KeyError – if object not found in any group

get_groups(uuids: list[str] | None = None) → list[ObjectGroup]

Return groups

add_group(title: str, group_uuid: str | None = None) → ObjectGroup

Add group to model

Parameters

- **title** – group title
- **group_uuid** – optional group UUID. If None, a new UUID is generated.

Returns
Created group object

get_object_group_id(obj: SignalObj | ImageObj) → str | None

Return group id of object

Parameters
obj – object to get group id from

Returns
group id or None if object is not in any group

get_group_object_ids(group_id: str) → list[str]

Return object ids in group

remove_group(group: ObjectGroup) → None

Remove group from model

add_object(obj: SignalObj | ImageObj, group_id: str) → None

Add object to model

add_objects(objects: Sequence[SignalObj | ImageObj], group_id: str) → bool

Add objects to the model atomically.

Parameters

- **objects** – Objects to add, in insertion order
- **group_id** – UUID of the target group

Returns

True if the short IDs of objects or groups already in the model changed; False when the objects were added after every existing object (the usual case), which leaves them unchanged.

Raises

- **KeyError** – If the target group does not exist
- **ValueError** – If an object UUID is duplicated or already exists

remove_object(obj: SignalObj | ImageObj) → None

Remove object from model

remove_objects(objects: Iterable[SignalObj | ImageObj]) → None

Remove objects from model, renumbering the remaining objects once

Removing many objects one by one (see *remove_object()*) would renumber all the objects of the model after each removal.

Parameters

objects – Objects to remove

Raises

KeyError – If an object is not in the model (nothing is removed)

get_object_from_number(number: int) → SignalObj | ImageObj

Return object from its number.

Parameters

number – object number (unique in model)

Returns

Object

Raises

IndexError – if object with number not found

get_objects(uuids: list[str]) → list[SignalObj | ImageObj]

Return objects with uuids

get_object_ids(flatten: bool = True) → list[str] | list[list[str]]

Return object ids, in order of appearance in groups

Parameters
flatten – if True, return a flat list of object ids, otherwise return a list of lists (one list per group)

Returns
List of object ids in order of appearance in groups.

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(flatten: bool = True) → list[str] | list[list[str]]

Return object titles, in order of appearance in groups

Parameters
flatten – if True, return a flat list of object titles, otherwise return a list of lists (one list per group)

Returns
List of object titles in order of appearance in groups.

get_object_from_title(title: str) → SignalObj | ImageObj

Return object with title.

Parameters
title – object title

Returns
object with title

Raises

- **KeyError** – if object with title not found
- **ValueError** – if multiple objects have the same title

get_title_reference_index() → dict[str, list[SignalObj | ImageObj | ObjectGroup]]

Return a dictionary mapping title references to groups and objects.

Each value is the list *find_by_title_reference()* returns for the reference. The index must be built again after objects or groups are added or removed.

reorder_groups(group_ids: list[str]) → None

Reorder groups.

Parameters
group_ids – list of group uuids

reorder_objects(obj_ids: dict[str, list[str]]) → None

Reorder objects in groups.

Parameters
obj_ids – dict of group uuids and list of object uuids

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: QTreeWidgetItem | None = None) → Iterator[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

resolve_title_reference(reference: str) → tuple[str, SignalObj | ImageObj | ObjectGroup] | None

Resolve a title reference to `(panel_str, obj)`.

Default implementation only looks up the tree's own model and returns an empty `panel_str`. Subclasses with access to several panels should override this method.

get_title_reference_display(reference: str) → str | None

Return the configured display text for a resolvable title reference.

get_source_title(reference: str) → str | None

Return the title of the object or group a reference resolves to.

build_title_reference_tooltip(text: str) → str

Return an HTML tooltip fragment listing the source objects referenced in `text`, or an empty string when no such reference is found.

handle_title_reference_click(reference: str) → None

Handle a click on a title-reference hyperlink. Default implementation selects the matching object within the tree's own model.

title_reference_at(pos: QPoint) → str | None

Return the title reference under viewport position `pos`, or `None` when the cursor is not over an anchor.

mousePressEvent(event: QMouseEvent) → None

Reimplement Qt method to handle short-ID hyperlink clicks.

mouseMoveEvent(event: QMouseEvent) → None

Reimplement Qt method to update the cursor over short-ID anchors.

hovered_index() → QModelIndex

Return the index of the item under the mouse cursor (invalid if none)

viewportEvent(event: QEvent) → bool

Reimplement Qt method to track the hovered item

changeEvent(event: QEvent) → None

Reimplement Qt method to keep hover events after a style change (some styles disable them when unpolishing item views)

populate_tree() → None

Populate tree with objects

update_tree() → None

Update tree

Note: If an item is not found in the tree, the tree is repopulated to ensure consistency between the tree and the model. This can happen in rare cases when objects are added to the model but the tree was not properly updated.

resize_title_column(items: Sequence[QTreeWidgetItem] | None = None) → None

Fit the title column to the titles.

Parameters

items – Items just added to a tree whose column was already fitted to the other items: the column is only widened if one of them needs it, without measuring every title again. All items when None.

add_group_item(group: ObjectGroup, resize: bool = True) → None

Add group item

Parameters

- **group** – Group to add, with its objects
- **resize** – If True, fit the title column to the new items

add_object_item(obj: SignalObj | ImageObj, group_id: str, set_current: bool = True) → None

Add item

add_object_items(objects: Sequence[SignalObj | ImageObj], group_id: str, set_current: bool = True) → None

Add multiple object items with a single tree update cycle.

Parameters

- **objects** – Objects to add, in insertion order
- **group_id** – UUID of the parent group
- **set_current** – If True, set the last object as current

update_item(uuid: str) → None

Update item

remove_item(oid: str, refresh: bool = True) → None

Remove item

remove_items(oids: Sequence[str], refresh: bool = True) → None

Remove items, in this order (looking the items up once for all)

Parameters

- **oids** – UUIDs of the objects or groups (unknown UUIDs are ignored)
- **refresh** – False to block the signals of the tree while removing

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

A delete button is shown at the right of the hovered item (signal, image or group), unless disabled (see *set_delete_button_enabled()*): clicking it emits SIG_DELETE_REQUESTED with the item UUID.

set_delete_button_enabled(enabled: bool) → None

Show or hide the delete button of the hovered item

Parameters
enabled – True to show the button

is_delete_button_enabled() → bool

Return True if the delete button of the hovered item is shown

delete_button_rect(index: QModelIndex) → QRect | None

Return the rectangle of the delete button of an item

The button is shown on the hovered item only, at the right edge of the visible part of the viewport (the title column may be wider than the viewport).

Parameters
index – model index of the item

Returns
Button rectangle in viewport coordinates, or None if the item shows no delete button.

get_delete_targets(uuid: str) → list[str]

Return the UUIDs of the items deleted by the delete button of an item

Parameters
uuid – UUID of the item whose delete button is clicked

Returns
UUIDs of the selected objects or groups if the item is part of the selection, else the item UUID only.

delete_button_tooltip(index: QModelIndex) → str

Return the tooltip of the delete button of an item

Parameters
index – model index of the item

Returns
Tooltip text

drawRow(painter: QPainter, option: QStyleOptionViewItem, index: QModelIndex) → None

Reimplement Qt method to draw the delete button of the hovered item

viewportEvent(event: QEvent) → bool

Reimplement Qt method to show the tooltip of the delete button and follow the mouse cursor over it

mousePressEvent(event: QMouseEvent) → None

Reimplement Qt method to press the delete button, without selecting or dragging the item

mouseMoveEvent(event: QMouseEvent) → None

Reimplement Qt method to follow the mouse cursor over the delete button

mouseReleaseEvent(event: QMouseEvent) → None

Reimplement Qt method to click the delete button

mouseDoubleClickEvent(event: QMouseEvent) → None

Reimplement Qt method to ignore double-clicks on the delete button (which would open the item in a separate view)

resolve_title_reference(reference: str) → tuple[str, SignalObj | ImageObj | ObjectGroup] | None

Resolve a title reference across the signal and image panels of the main window, so titles can reference objects living in either panel.

handle_title_reference_click(reference: str) → None

Select the referenced object, switching active panel if needed.

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

get_current_object_uuid() → str | None

Return current object uuid, or None if current item is a group or if no item is current

set_current_object(obj: SignalObj | ImageObj) → None

Set current object

select_uuids(uuids: Sequence[str], notify: bool = True) → None

Select objects or groups atomically, the last one becoming current

Parameters

- **uuids** – UUIDs of the objects or groups to select
- **notify** – if True, publish the complete final state once (see *item_selection_changed()*)

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

update_automatic_styles() → None

Give the visible curves the automatic style of their position

Visible curves are numbered in the order of their signals in the panel. Each curve whose color and line style were not chosen by the user is given the automatic style of its position (see `CurveStyles.style_at()`), so that the first curves shown are always given the first styles.

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

remove_items(oids: Iterable[str]) → None

Remove the plot items associated to objects, at once

Removing the items one by one would refresh the plot observers (e.g. the item list panel) once per item: removing many items would take a long time.

Parameters
oids – Object uuids (objects without plot item are ignored, see *remove_item()*)

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_existing_items() → None

Update all existing plot items from their objects and current settings

Unlike *refresh_plot()*, the visibility of the plot items and the shape items are left unchanged. This applies a change of display settings (e.g. signal auto downsampling or line width) to the plot items of all objects, and not only to those of the selected objects.

update_item_on_plot(oid: str, just_show: bool = False) → None

Update plot item.

Parameters

- **oid** – object uuid
- **just_show** – if True, only show the item (do not update it). Defaults to False.

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

remove_items(oids: Iterable[str]) → None

Remove the plot items associated to objects, at once

Removing the items one by one would refresh the plot observers (e.g. the item list panel) once per item: removing many items would take a long time.

Parameters
oids – Object uuids (objects without plot item are ignored, see *remove_item()*)

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_automatic_styles() → None

Update the automatic styles of the visible plot items

This base implementation does nothing.

update_existing_items() → None

Update all existing plot items from their objects and current settings

Unlike *refresh_plot()*, the visibility of the plot items and the shape items are left unchanged. This applies a change of display settings (e.g. signal auto downsampling or line width) to the plot items of all objects, and not only to those of the selected objects.

update_item_on_plot(oid: str, just_show: bool = False) → None

Update plot item.

Parameters

- **oid** – object uuid
- **just_show** – if True, only show the item (do not update it). Defaults to False.

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, source_panel: BaseDataPanel | 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, source_panel: BaseDataPanel | 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

apply_shared_contrast(zmin: float, zmax: float) → None

Apply a contrast change coming from the image panel.

plot_lut_changed(plot: BasePlot) → None

Synchronize ROI editor contrast changes with the source image panel.

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

Extends the SigimaX plot widget with the "Process signal" action, which sends a profile to the DataLab signal panel.

Parameters

plot_type – 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

PLOTWIDGET_CLASS

alias of *DataLabPlotWidget*

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

append_informed

True when the user was already told, for the current opening, that the provenance of appended workspaces is not imported (no repeat).

cancelled_files: list[str]

Files whose appending the user cancelled (see `__confirm_append()`)

save_file(filename: str) → None

Save all signals and images from DataLab model into a HDF5 file

The file is replaced only once completely written: if saving fails, the previous file (if any) is left untouched.

open_file_headless(filename: str, reset_all: bool) → bool

Open native DataLab HDF5 file without any GUI elements.

This method can be safely called from any thread (e.g., the console thread) as it does not create any Qt widgets or dialogs. When a file holding provenance is appended (reset_all False), its provenance is not imported and the notice `datalab_capsule.recorder.APPEND_NOTICE` is added to `mainwindow.provenance.notices` (and logged).

Parameters

- **filename** – HDF5 filename
- **reset_all** – Reset all application data before importing

Returns
True if file was successfully opened as a native DataLab file, False if the file format is not compatible (KeyError was raised)

Raises
ProvenanceFormatError – If the provenance block is invalid (raised before the workspace is replaced).

open_file(filename: str, import_all: bool, reset_all: bool) → None

Open HDF5 file

When the file holds provenance and is appended to the current workspace, the user is warned before anything changes (see `__confirm_append()`) and may cancel. A file that is not a DataLab workspace is imported as a generic HDF5 file.

open_workspace_file(filename: str) → None

Open a DataLab workspace file, replacing the workspace.

Unlike *open_file()*, a file that is not a DataLab workspace is never imported as a generic HDF5 file.

Raises

- **ValueError** – If the file is not a DataLab workspace.
- **ProvenanceFormatError** – If the provenance block is invalid.

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

FOUR

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.

4.1 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

4.2 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

4.3 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

Writing a plugin is the easiest way to add a feature to DataLab: a plugin may be a single Python file, and publishing it in the DataLab plugin catalog makes it available to all DataLab users, without any change to DataLab itself.

See the tutorial From an idea to the plugin catalog, and the contribution guide of the DataLab plugin catalog.

4.4 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.

4.4.1 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.

Maintainer tasks

A few maintenance tasks are intentionally not run by the release CI and must be performed locally on a Windows workstation by a project maintainer. They produce committed assets that the CI then consumes as-is.

Refreshing documentation screenshots

The PNG files under `doc/images/` are versioned. The screenshot generator renders actual Qt windows, dialogs and menus offscreen: it does not capture the desktop, drive the mouse or require Computer Use. Run it from a source checkout with the development environment configured in `.env`, including Sigima's test data, Qt and OpenCV. The reference captures are generated on Windows with the Segoe UI font.

The screenshot pipeline produces per-language files (`foo.fr.png` / `foo.en.png`) thanks to Sphinx's `figure_language_filename` configuration. RST sources reference the unsuffixed name (`foo.png`); each language build automatically picks the matching suffix when available and falls back to the unsuffixed file otherwise.

Re-run this whenever a change affects the rendered UI (menus, dialogs, panels, themes):

```
scripts\update_screenshots.bat
```

or via the 🖼 Refresh doc screenshots VS Code task. Each language runs in its own process with a disposable home directory and in-memory default settings: personal preferences and installed plugins are not changed. The theme, scale, window sizes and tutorial data are fixed. Native window decorations are intentionally absent; popup menus are rendered by Qt and composited over their parent window.

To update only the interface and a tutorial, or render previews outside the source tree:

```
scripts\update_screenshots.bat --group interface spectrum
scripts\update_screenshots.bat --group blobs --language en --output-dir build\
↪screenshot-preview
```

Available groups are `interface` (startup, settings and History), `gallery` (the existing demonstration scenes), `metadata` (signal and image metadata dialogs), `fabry-perot`, `spectrum` and `blobs`. All groups and both languages are selected by default. Tutorial figures remain shared English PNGs under `doc/images/tutorials/`; interface figures are localized under `doc/images/shots/`. The blob tutorial executes the example plugin with seeded random data; its plugin-directory figure uses illustrative paths rather than personal or temporary directories.

The scenarios are in `datalab/tests/scenarios/tutorial_screenshots.py`. They check that annotations are stored separately from metadata, peak detection creates XY markers rather than another signal, and the blob detector's color filter is disabled. Generation stops on errors; the focused regression tests also check PNG output and protection of an existing configuration.

Inspect the resulting figures and build the English and French documentation before committing the updated PNGs. Once a `foo.fr.png` + `foo.en.png` pair exists for a given asset, the unsuffixed `foo.png` becomes a pure fallback and can be removed in a follow-up cleanup commit.

Regenerating graphical resources

The application icon (`resources/DataLab.ico`) and the WiX installer bitmaps (`wix/dialog.bmp`, `wix/banner.bmp`) are generated from SVG sources. Re-run only when the SVG sources change:

```
scripts\build_resources.bat
```

or via the 🎨 Generate resources VS Code task.

The images displayed by the application (splash screen, logos, plot watermark) are generated in `datalab/data/logo/` together with their `@2x` high-resolution variants, used on high-DPI displays: to regenerate only these images, run `scripts\build_resources.bat logo`.

4.4.2 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 88 characters (the line length configured for `ruff` in `pyproject.toml`).
- 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 "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 "Pylint" to run pylint on the project.

> **Note**
>
> A pylint rating greater than 9/10 is expected to merge a pull request. Unlike ruff, pylint is not run by the continuous integration: check the rating before submitting your 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.

4.4.3 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. A release branch is created only when the maintenance and development lines must diverge.

Branching Model

Main Branches

- `main`: Represents the stable, production-ready version of the project.
- `develop`: Prepares the next release, i.e. a patch release (e.g., 1.0.1) as long as it only contains fixes since the last release, or a minor or major release (e.g., 1.1.0) once new features are merged.

Feature Branches

- `feature/feature_name`: Used for the development of new features.
  - Created from `develop`.
  - Merged back into `develop` once completed.
  - Deleted after merging.

Release Branch

- `release`: Represents the current maintenance line for patch releases, when `develop` already contains changes for the next minor or major release.
  - Created from `main` only when a patch release is needed while `develop` contains changes that must not be released yet.
  - Accepts `fix/xxx` and `hotfix/xxx` branch merges.
  - Merged back into `main` for each patch release tag (e.g., 1.0.1, 1.0.2).
  - Merged into `develop` after each patch release, so that its fixes reach `develop` without cherry-picking.
  - Deleted when the next minor or major version is released.

> **Note**
>
> The `release` branch is not versioned (no `release/1.0.x`), and it does not always exist. Without `release` branch, `develop` prepares the next patch release: fixes and patch releases are made from `develop`. When a new minor version is released (e.g., 1.2.0), the `release` branch is deleted; a new one is created from the fresh tag only if a patch for 1.2.1 is needed while `develop` already contains changes for 1.3.0.

Bug Fix Branches

- `fix/xxx`: Used for general bug fixes that are not urgent.
  - Created from `develop`, or from `release` when this branch exists and the fix must be in the next patch release.
  - Merged back into the originating branch once completed.
  - Deleted after merging.
- `hotfix/xxx`: Used for urgent production-critical fixes.
  - Created from `release` (if exists) or `develop` (if no `release` branch).
  - Merged back into `release` or `develop`.
  - A fix merged into `release` reaches `develop` when `release` is merged into `develop`.
  - Deleted after merging.

> **Note**
>
> Hotfixes (high-priority fixes) will be integrated in the next patch release (X.Y.Z -> Z+1). Fixes (low-priority fixes) will be integrated in the next release prepared by `develop`: a patch release, or a feature release (X.Y -> Y+1) once `develop` contains new features.

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 feature/feature_name
```

2. Develop the feature and commit changes.
3. Merge the feature branch back into `develop`:

```
git checkout develop
git merge --no-ff feature/feature_name
```

4. Delete the feature branch:

```
git branch -d feature/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

For the next release prepared by `develop` (target: `develop`):

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
```

For the next patch release, when a `release` branch exists (target: `release`):

1. Create a bug fix branch from `release`:

```
git checkout release
git checkout -b fix/bug_description
```

2. Apply the fix and commit changes.
3. Merge the fix branch back into `release`:

```
git checkout release
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 `feature/feature_name` branch. Always branch from `develop` or `release` to ensure fixes are correctly propagated.

```
# Incorrect:
git checkout feature/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 `release` (or `develop` if no `release` branch exists):

```
git checkout release  # or: git checkout develop
git checkout -b hotfix/critical_bug
```

2. Apply the fix and commit changes.
3. Merge the fix back into `release` (or `develop`):

```
git checkout release  # or: git checkout develop
git merge --no-ff hotfix/critical_bug
```

4. If the fix was merged into `release`, merge `release` into `develop`, at the latest after the patch release:

```
git checkout develop
git merge --no-ff release
```

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 `release` is merged into `develop` to avoid losing fixes in future releases.

Workflow for Patch Releases

When ready to release a patch version (e.g., 1.0.1, 1.0.2):

If no `release` branch exists, `develop` only contains fixes since the last release: release it with the workflow for minor/major releases below, using the patch version number. Otherwise, release the `release` branch:

1. Ensure the `release` branch contains all desired fixes.
2. Merge `release` into `main`:

```
git checkout main
git merge --no-ff release
```

3. Tag the release on `main`:

```
git tag -a v1.0.1 -m "Release version 1.0.1"
git push origin main --tags
```

4. Pushing the tag starts the Release workflow, which checks the tag, runs the tests and builds the release files, then creates a draft GitHub release. Review the draft release and publish it: this uploads the package to PyPI and deploys the documentation. Never create a GitHub release by hand (see `RELEASING.md` at the root of the repository).
5. Merge `release` into `develop`, and keep the `release` branch for additional patches in the same minor version series:

```
git checkout develop
git merge --no-ff release
```

Workflow for Minor/Major Releases

When ready to release a new minor or major version (e.g., 1.1.0, 2.0.0):

1. Merge `develop` into `main`:

```
git checkout main
git merge --no-ff develop
```

2. Tag the release on `main`:

```
git tag -a v1.1.0 -m "Release version 1.1.0"
git push origin main --tags
```

3. Pushing the tag starts the Release workflow, which checks the tag, runs the tests and builds the release files, then creates a draft GitHub release. Review the draft release and publish it: this uploads the package to PyPI and deploys the documentation. Never create a GitHub release by hand (see `RELEASING.md` at the root of the repository).

4. Delete the old `release` branch (if exists):

```
git branch -d release
git push origin --delete release
```

5. Create a new `release` branch from `main` only when a patch (e.g., 1.1.1) is needed while `develop` already contains changes for the next minor or major release:

```
git checkout main
git checkout -b release
git push -u origin release
```

Best Practices

- Regularly rebase feature branches on `develop` to stay up to date:

```
git checkout feature/feature_name
git rebase develop
```

- Avoid long-lived branches to minimize merge conflicts.
- Merge `release` into `develop` after each patch release: fixes reach `develop` without cherry-picking.
- While `develop` prepares a patch release (no `release` branch), keep changes for the next minor or major release in their branches until the patch release, or create the `release` branch first.
- Clearly differentiate between `fix/xxx` (non-urgent fixes) and `hotfix/xxx` (critical production fixes).
- When creating the `release` branch, update release notes to indicate which version it targets (e.g., add a comment in the merge commit: "Create release branch for v1.0.x maintenance").
- When it exists, the `release` branch represents the current maintenance line. To know which version it targets, check the most recent tag on `main` or the release notes.

Takeaway

This workflow ensures a structured yet flexible development process while keeping `main` stable and `develop` always updated with the latest changes.

When needed, the `release` branch provides a dedicated maintenance line for patch releases, allowing `develop` to proceed with new features without interference. This solves the problem of coordinating unreleased changes across the PlotPyStack ecosystem (DataLab, SigimaX, Sigima, PlotPy, guidata, PythonQwt) by providing a stable branch for CI testing during maintenance cycles.

4.4.4 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.
- exe: building the stand-alone Windows executable with PyInstaller (pinned version), including OpenCV and PyQt5.
- 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.

4.4.5 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.

⚠ DataLab relies on its sibling repositories Sigima (processing features), SigimaX (application framework) and DataLab-Capsule (workspace capsules). If your contribution needs unreleased changes of these repositories (e.g. a processing feature, implemented in Sigima), you will need to work on them too. Either install them in editable mode:

```
pip install -e ../Sigima -e ../SigimaX -e ../DataLab-Capsule  # Adjust the paths to␣
↪your local repositories
```

or add them to the `PYTHONPATH` variable of the `.env` file (see below), e.g. `PYTHONPATH=.;../Sigima;../SigimaX;../DataLab-Capsule/src`.

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
3.10	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
3.14	OK	3.14.5.0

⚠ 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 DATALAB_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 official Windows executable and MSI installer are built by the GitHub CI release pipeline. The Python version embedded in the standalone application shipped to end users is defined by the `python-version` input default of the reusable build workflow (`.github/workflows/_build.yml`), consumed by its `build-msi` job; the release workflows do not override it. That workflow is the single source of truth — refer to it for the exact version currently in use.

The Windows installer is built using WiX Toolset V4 (the CI release pipeline installs the latest 4.x version, with the WiX UI extension 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.*
wix extension add -g WixToolset.UI.wixext/4.0.5
```

First, you need to generate the installer script from a generic template (replace `X.Y.Z` by the version of DataLab, e.g. `1.4.0`):

```
python wix/makewxs.py DataLab X.Y.Z
```

Building the installer is done using the following command:

```
wix build wix\DataLab-X.Y.Z.wxs -ext WixToolset.UI.wixext
```

Alternatively, `scripts\build_installer.bat` (Visual Studio Code task "□ Create installer") runs both commands for the current version of DataLab.

Release smoke test

Before uploading the MSI installer, the `build-msi` job of the reusable build workflow smoke-tests it with `scripts/frozen_smoke_test.py`: it installs the MSI in a disposable folder, compares the installed files with the PyInstaller folder, starts the installed `DataLab.exe` in unattended mode with screenshots, runs `DataLab.exe --smoke-test` (installation check implemented in `datalab/utils/smoketest.py`), uninstalls the MSI and checks that the release artifacts were not modified. Any failure fails the job, so nothing is published. The screenshots, reports and logs are uploaded as the `msi-smoke-test` artifact.

To run it locally, after building the executable and the installer (the script refuses to run if DataLab of the same major version is installed on the computer, as it uninstalls the MSI at the end):

```
python scripts/frozen_smoke_test.py all --msi wix/DataLab-1.4.0.msi --dist dist/
↪DataLab --workdir build/msi-smoke-test
```

The `gui` and `probe` commands of this script only start an executable (e.g. `dist/DataLab/DataLab.exe`) with screenshots, or run its installation check.

When DataLab requires versions of Sigima, SigimaX or DataLab-Capsule which are not published yet, run the "Release (dry-run)" workflow with the `dependencies` input set to `development`: these packages are then built from their branch named like the dispatched branch, or from `develop`.

Third-party Software

The following software may be required for maintaining the project:

Software	Description
gettext	Translations
Git	Version control system
MikTeX	LaTeX distribution on Windows

4.4.6 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 included in the 1.0 release (November 2025):• The goal was to decouple the data model, computation, and I/O from the UI• This led to the creation of the Sigima library, which now handles data processing independently of DataLab's GUI• This redesign improves maintainability, testability, and extensibility of the codebase• This also enables DataLab to be used as a library (through Sigima) in other software projects, fostering integration and reuse

🏗 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 (started)• Support for time series• Database connectors• Spyder plugin for interactive data analysis
🛠 Maintenance	• Transition to gRPC for remote control• Drop Qt5 support and migrate to Qt6

🗘 Support for Data Acquisition 🚧 Started

ⓘ Data acquisition has started with DataLab 1.4: an application plugin may provide instruments, opened in a window with a live view, their settings and an Acquire button adding the acquired signals or images to the workspace (see Application plugins). The TANGO Instruments demonstration plugin acquires the attributes of TANGO devices this way, and the camera and pulse characterization plugins of the DataLab plugin catalog provide simulated instruments. The original exploration notes are kept below for context.

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 ✓ Delivered

A web frontend has been delivered as DataLab-Web, the browser-native edition of the platform (see the ecosystem overview). Rather than running on a server, it runs the Sigima computation engine entirely inside the browser — CPython compiled to WebAssembly via Pyodide — paired with a React user interface. This keeps DataLab a local application: no server, no account, and the data never leaves the user's machine, consistent with the project's privacy-first philosophy. Try it at datalab-platform.com/web.

⏱ 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 Partially Delivered

> The first feature below has been delivered by DataLab-Kernel (see its documentation and the ecosystem overview): a Jupyter kernel giving notebook access to DataLab workspaces, with optional live synchronization with a running DataLab instance. The original notes are kept below for context.

Although DataLab can already be remotely controlled from a Jupyter notebook — thanks to its existing remote control capabilities (see Remote controlling (XML-RPC)) — 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 (XML-RPC)), 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 ✓ Delivered

> A Jupyter kernel has been delivered as DataLab-Kernel (see its documentation and the ecosystem overview), providing notebook-driven access to DataLab workspaces with optional live synchronization to the desktop GUI. The original exploration notes are kept below for context.

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 (XML-RPC)), 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.

⦸ Drop Qt5 Support and Migrate to Qt6 🚧 Started

> ⓘ DataLab supports PyQt6 since version 1.1 (tested by the continuous integration), but PyQt5 remains the reference Qt binding: it is installed by the `qt` extra and used by the stand-alone Windows installer.

With the end-of-life of Qt5 in 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 ✓ Delivered

> ⓘ Delivered with DataLab 1.4: the `datalab-plugin create` command creates a ready-to-use plugin in a single Python file or, with `--project`, a layered installable project with tests and a release workflow, ready to publish in the DataLab plugin catalog (see Developing plugins and the tutorial From an idea to the plugin catalog). The original notes are kept below for context.

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 1.4, 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
1.420	• Welcome page with application tiles, and Applications panel• Application plugins: headless recipes, packaged examples, tools and instruments• DataLab plugin catalog: official and community plugins installed from DataLab, local and offline catalogs• Plugin generator: `datalab-plugin create`• Workspace provenance and workspace capsules• Live processing preview, brightness and contrast, RAW image import, NeXus files
1.320	• History panel• Built-in AI assistant and command palette• Spectral analysis support• Regions of interest from detection (auto mask)• DataLab now builds on the SigimaX application framework
1.220	• Plugin configuration dialog and plugin hot-reload• Multi-instance detection• Remote control API: push modified objects back to DataLab
1.120	• Web API for HTTP/JSON integration, used by DataLab-Kernel• PyQt6 compatibility• Remote control API enhancements
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
0.10212	• 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
0.9201 1	• 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

FIVE

RELEASE NOTES

This section contains the release notes for all versions of DataLab, documenting new features, improvements, bug fixes, and breaking changes.

5.1 Version 1.4

5.1.1 DataLab Version 1.4.1

Requirements

- Runtime requirements are unchanged since version 1.4.0.
- Third-party:
  - PyYAML is now part of the `test` optional dependencies: the test suite needs it to check the projects created by the plugin generator.

🛠 Bug Fixes since version 1.4.0

- Tutorial examples and metadata: The custom denoising example no longer produces undefined values on constant image regions, and the Spyder tutorial no longer produces undefined values on grids containing the first peak position. The metadata paste dialog now explains that annotations are copied and pasted separately (fixes Issue #567).
- Scientific processing (⚠ results change): With Sigima 1.4.1 or later, X–Y mode uses only the signals' common interval instead of extending one signal beyond its data, while retaining the interpolation confirmation; disjoint domains are refused. The enclosing circle now contains the selected pixel centers in physical coordinates, quadratic interpolation passes through the samples, pulse offsets retain their sign, and flat-field correction refuses zero denominators on corrected pixels. Previously recorded calls of these operations are refused at replay rather than silently giving different results; existing stored data is unchanged (fixes Issue #565 and Sigima Issue #130).
- Integer images and X–Y mode (⚠ results change): with Sigima 1.4.1 or later, on signed integer images whose range of values exceeds the capacity of their type (e.g. `int16` values from -30000 to +30000), Analysis > Minimum enclosing circle center no longer gives a much too large circle, Analysis > Contour detection finds the contours, and Analysis > 2D peak detection no longer reports spurious peaks. Flat-field correction of an integer image now gives a floating-point image instead of wrapping the values exceeding the range of the image type. With a region of interest on the first signal, X-Y mode and Replace X by other signal's Y now keep only the points within this region, instead of mixing the original coordinates of the other points into the result. X-Y mode results now take the titles and units of the Y axes of the two signals, signals with different X units are refused, and signals sharing the same X grid are paired without interpolation. Previously recorded calls of the changed operations are refused at replay (fixes Issue #571 and Sigima Issue #136).
- Plugin installation: installing a plugin whose required packages are missing now names these packages and tells how to provide them, instead of reporting an unexplained "Dependency wheelhouse not found" error with a temporary folder.

- Brightness and contrast: the Minimum and Maximum fields of Processing > Exposure > Brightness and contrast (in its dialog and in the Processing tab of the properties panel) showed numbers with 17 significant digits in fields too narrow for them, so that their beginning was cut off, e.g. `835736228608e-43` instead of `6.5127532828723551e-44`. These fields now show the complete value, with a precision matching that of the sliders (e.g. `0` and `1.7557`), and the exact value in their tooltip; a value which is not modified is kept exactly (fixes Issue #510).
- Processing parameters: moving the slider of a numeric parameter gave values such as `0.30000000000000004` or `1.2000000000000002` instead of `0.3` or `1.2`, which were then used and recorded as processing parameters, e.g. with Sigmoid correction and Adaptive histogram equalization (Processing > Exposure), the Butterworth filter (Processing > Frequency filters) or the signal Peak detection (Analysis). Slider values are now free of these spurious digits (fixes Issue #510).
- Image coordinates: since DataLab 1.2.0, Set uniform coordinates was missing from Processing > Axis transformation in the Image panel, so this feature could no longer be run from the menus. It is available again, next to Polynomial calibration (fixes Issue #515).
- Web API: starting the Web API server through the remote control (XML-RPC), e.g. with `RemoteProxy.start_webapi_server()`, used port 8080 instead of the automatic port (18080, or the next free one), so that DataLab-Kernel could not find the server automatically. Unless a port is given, the server now uses the automatic port, as when it is started from File > Web API (fixes Issue #516).
- AI Assistant: the Enable AI Assistant option of the settings had no effect: the assistant remained available, although the documentation stated that no request was sent when it was disabled. This option is removed: the assistant contacts its provider only when you send a message or click Test connection, as the documentation now states. A new Settings… button of the AI Assistant panel opens the AI Assistant tab of the settings, and the messages of the panel point to File > Settings instead of a nonexistent Edit > Preferences menu (fixes Issue #571).
- Visualization settings: the Downsampling max points field could remain editable when the settings dialog opened with Use auto downsampling unchecked, e.g. after checking this option and cancelling the dialog. It now follows the checkbox as soon as the dialog opens (fixes Issue #571).
- Stability analysis: Overlapping Allan variance was missing from Processing > Stability analysis in the Signal panel, so it could only be computed with the other methods, through All stability features…. It is now listed with the other variances (fixes Issue #518).
- Gaussian bandpass filter: the Inverse FFT result parameter of Processing > Frequency filters > Gaussian bandpass had no effect: Absolute value gave the same image as Real part, although the parameter was recorded in the provenance as if it had been applied. With Sigima 1.4.1 or later, Absolute value gives the modulus of the inverse FFT, and the title of the result shows the chosen option; Real part, the default, gives the same result as before. Steps of this filter recorded with an earlier version are refused when they are verified or replayed, instead of giving another result (fixes Sigima Issue #123).
- Cardinal sine signal: the cardinal sine of Create > Cardinal sine, described as `y = y0 + A sinc(2π f x + φ)`, used the normalized definition `sinc(u) = sin(πu)/(πu)`: with a frequency of 1 Hz, its first zeros were at about ±0.159 instead of ±0.5. With Sigima 1.4.1 or later, it uses the definition `sinc(u) = sin(u)/u` (with `sinc(0) = 1`), as the 2D sinc image, which is unchanged; the creation dialogs of both now show this definition. For the same parameters, a cardinal sine signal created, edited or recomputed from the History panel is now different; signals already created, e.g. in a saved workspace, keep their data (fixes Sigima Issue #60).
- Web API: when required components could not be imported, the menu and remote control suggested installing a nonexistent extra. They now name the required dependencies and explain how to check or repair the installation using its original method or package manager; pip users can run `python -m pip check` to check dependency requirements (fixes Issue #525).
- Web API: each object added through the Web API, e.g. from DataLab-Kernel or a Jupyter notebook, showed "Remote request failed: Object '…' not found" in the status bar, although the object was added. Checking that an object name is free is no longer reported as a failed request (fixes Issue #528).

- Web API: when replacing an existing workspace object, a failed addition of the replacement could delete the original object. The original object now remains available if the replacement cannot be added (fixes Issue #561).
- Deleting objects: deleting many objects at once, or clearing the workspace, froze DataLab for a long time when many curves were displayed (e.g. about 20 seconds for 600 signals). It now takes about one second (fixes Issue #528).
- Plugins: after running a processing feature of a plugin whose computation functions are not in the module of its plugin class, such as the plugins created with `datalab-plugin create --project`, saving the workspace to an HDF5 file failed with "cannot serialize None value in dict". These results are now saved, with the name and version of their plugin (fixes Issue #543).
- Applications panel: when application plugins were installed, docking the floating Applications panel back into the main window could crash DataLab. The panel can now be docked and undocked safely (fixes Issue #527).
- Plugin testing: scripts and tests using `datalab_test_app_context`, e.g. to test a plugin, sometimes crashed when the Python interpreter exited, after their checks had passed. Their main windows are now destroyed before the Qt application, when the interpreter exits (fixes Issue #556).
- Development tooling: the VS Code tasks of the source repository reported success even when their command failed (e.g. a failed test, a linting error or a documentation build error), and the tasks chained after them still ran, e.g. updating the translation catalogs after a failed documentation build. These tasks now fail with their command; Coverage full still produces the coverage report when tests fail (fixes Issue #552).

📖 Documentation since version 1.4.0

- Corrected installation and remote-control instructions, descriptions of History and plugins, and tutorial screenshots so they reflect the current application (fixes Issue #569).
- Corrected the description of the pulse amplitude, which never falls back on the peak-to-peak amplitude, and documented the bias of the default FWHM method on smooth peaks, the unit of the phase of periodic signals (degrees in the dialogs), the effect of hot pixels on the enclosing circle, of near-zero values on the flat-field correction, how the Web API replaces an object, and the cases where DataLab and DataLab-Web results can be compared. The laser beam tutorial now gives the centroid row used by its screenshots (fixes Issue #571).
- Updated the Fabry-Pérot, spectrum and blob-detection tutorial screenshots, the startup page, settings and History panel to match the current interface. Maintainers can regenerate these figures with a single Qt-based script, without manual desktop captures or changes to personal preferences (fixes Issue #520).
- Translated card titles in the French documentation now show the same icons as their English counterparts, on the home page and the main section pages (fixes Issue #536).
- The DataLab plugin catalog is now presented on the home page, in the navigation bar and in the pages describing the platform, its key features and how to contribute
- The plugin documentation is reorganized into an overview page and dedicated pages for installing and managing plugins, deploying them in an organization, developing them, application plugins and the public API; the development page starts with a quick start, "Your first plugin in five minutes", and presents the plugins of the catalog as examples
- New tutorial, "From an idea to the plugin catalog": create a plugin in a single Python file, try it, then publish it in the DataLab plugin catalog, without any PyPI account
- New use case pages present the characterization of scientific cameras, the analysis of transient and pulsed signals, and instrument control and data acquisition, with the Camera & Detector Characterization, Pulse & Transient Characterization and TANGO Instruments plugins of the DataLab plugin catalog; the non-destructive testing page now presents the X-ray Radiography & NDT plugin, also cited as an example in the plugin documentation (Issue #542)
- The roadmap is updated: the plugin template is delivered, data acquisition has started, and the milestones of versions 1.1 to 1.4 are listed

- The documentation is corrected where it no longer matched the application (Issue #519): menu paths and labels of the tutorials and feature pages, settings and their default values, formulas and results of several processing and analysis features, supported file formats, annotations (stored apart from metadata), the Web API, whose token is not required for local requests by default, the remote control, limited to the local machine, plugins targeting DataLab-Web, macros, installation commands (the conda-forge package is `datalab`; `[qt]` and `[opencv]` extras), and several examples that no longer worked
- On the home page and the Getting started page, the logos of the funders and the icons of the three operating modes are shown again at their intended size, without a white background in the dark theme, and the logos of CEA, CODRA and NLnet link again to their websites instead of opening the image (fixes Issue #521)
- The Web API documentation and the tooltip of the Web API localhost bypass setting now warn that, while the Web API server is running with this setting enabled (default), any web page open in the browser, and not only local applications, can access the workspace without a token (Issue #522)

🔧 Improvements since version 1.4.0

- Plugin generator: the release workflow of a generated plugin project (`.github/workflows/release.yml`) now checks that the tag matches the version of both `pyproject.toml` and the package, runs the tests before building the wheel, and publishes the wheel with its SHA-256 checksum and its build provenance attestation, which users can check with `gh attestation verify`. A tag such as `v0.2.0-rc1` publishes a release candidate as a GitHub pre-release, without plugin catalog entry. The generated test no longer compares the plugin version with a fixed value, so it no longer fails when the version changes. Existing plugin projects are not affected.
- Plugin catalog: a plugin may now require packages that users install themselves in DataLab's Python environment, such as PyTango for the TANGO Instruments plugin. The Catalog tab shows these packages and offers to install the plugin only once they are installed; in the standalone application, such plugins are shown as not available.
- Welcome page: the Applications section is now shown even when no application is installed: it then invites you to discover plugins, and "Discover plugins…" opens the Catalog tab of the plugin configuration dialog, to find and install the plugins of the DataLab plugin catalog; once applications are installed, a "Discover plugins…" button of the section keeps this access. The online documentation and the release notes are now links at the bottom of the page, so that the Walkthroughs column only holds the AI assistant, the guided tour and the demo.
- Welcome page: application tiles no longer grow with their description, which made the page crowded once several application plugins were installed (each with a tile for its examples). Tiles now all have the same height and share the width of the page, from the left of Quick actions to the right of Walkthroughs, with as many tiles per row as the width allows: the title stands next to the icon, the description follows it directly, and both are limited (two lines for the title, three for the description), ending with "…" when they are longer, the full text being shown in a tooltip (closes Issue #545).
- Plugin instruments: the window of a plugin instrument, e.g. the camera, oscilloscope or radiography simulators, has a new Restore defaults button, which sets the settings of the instrument back to their default values and refreshes the view (closes Issue #545).
- Plugin tools: a plugin may declare a tool with `menu=False` to place it in one of its own submenus: DataLab then lists it in the Applications panel only, instead of also adding it at the end of the plugin's submenu (closes Issue #545).

5.1.2 DataLab Version 1.4.0

Requirements

- DataLab-Platform:
  - Sigima 1.4.0 or later is required: it provides operation contracts and replay qualification, RAW image import, format-aware image export, portable annotations, experimental-source metadata, NPZ object serialization and pixel-center ROI rules.
  - SigimaX 1.1.0 or later is required: it provides portable PlotPy annotation adapters, high-contrast curve styles, high-DPI splash screen and watermark rendering, and NeXus / multidimensional HDF5 browsing.
  - DataLab-Capsule 0.2.x (`>= 0.2.0, < 0.3`) is required for workspace provenance recording, verification and `.dlcapsule` exchange.
- PlotPyStack:
  - guidata 3.17.2 or later is required: it provides histogram-backed range items, module-qualified icon precedence, high-DPI pixmap helpers, and gives every window a title bar following the color theme on Windows, including the windows of plugins (closes Issue #492).
  - PlotPy 2.12.1 or later is required: it provides sharp SVG icons on high-DPI displays, min/max signal auto-downsampling and faster image overlay interactions.
- Third-party:
  - The Python package and newly generated plugin projects now require Python 3.10 or later. Python 3.9 users must keep an earlier compatible release. The standalone application continues to include its own Python interpreter.
  - The Windows test runner excludes Python 3.9 from current matrices, while retaining explicit tests of pinned historical releases.
  - pip 22.2 or later is required to resolve and install additional plugin wheel dependencies.
  - NumPy up to 2.5 (`>= 1.22, < 2.6`) and SciPy up to 1.18 (`>= 1.10.1, < 1.19`) are now supported.

✧ New Features

Welcome page:

- A new Welcome page, similar to the one of DataLab-Web, is shown at startup and whenever the current signal or image panel is empty. It gathers the main actions to get started: create a signal or an image, open files, browse an HDF5 file, open an HDF5 workspace, import text data, ask the AI Assistant, take the guided tour, run the demo, read the documentation or browse the release notes
- The page is a dockable panel, tabbed with the signal and image views: it remains available after data is created or opened, and can be reopened at any time from the ? menu (Welcome page) or from the View menu
- A Show welcome page when the current panel is empty option, available on the page itself and in the General settings, controls whether the page is shown automatically, including at startup

Brightness and contrast:

- A new Processing > Exposure > Brightness and contrast operation provides a histogram with synchronized minimum, maximum, brightness and contrast controls, deterministic Auto and Reset actions, ROI-aware remapping, and live preview. One input window is initialized from the first selected image and applied to the full selection, with each result preserving its source image and data type.
- Intensity windows may extend beyond the observed data range. Exact floating-point bounds survive editing and reapplication, while declared parameter constraints remain enforced and Cancel preserves the original parameters.

RAW image import:

- Headerless RAW image files (`*.raw`) can now be opened from File > Open…, File > Open from directory…, drag-and-drop, the AI Assistant (`load_file` tool) and remote control (`load_from_files` and `load_from_directory` with an optional `RawImageImportParam` argument) (implements Issue #237)
- The RAW image import dialog defines the pixel data type, image width and height, byte offset to the first image, number of images, byte gap between consecutive images, byte order, and the White is zero option (which displays the imported image with an inverted gray colormap without modifying pixel values). Multi-image RAW files create one image per frame
- When several RAW files are imported at once, DataLab asks whether to apply the same import parameters to all of them or to set the parameters of each file (each dialog starting from the values confirmed for the previous file); recorded History actions store the import parameters so that replaying them does not reopen the dialog

Format-aware image export:

- Saving an image with File > Save… now opens an Export image dialog adapted to the selected file format, with a live preview showing the image in gray levels as it will be read back from the exported file (including quantization and lossy compression for BMP, PNG, JPEG, JPEG 2000 and TIFF files) (implements Issue #256)
- Data preparation covers normalization (none, min-max, percentile or manual bounds), out-of-range behavior (clip or rescale), non-finite value policy (error, clip or replace), gamma correction, inversion and conversion to a data type supported by the destination format; when exporting a floating-point or unsupported image to an integer-only format (such as PNG, JPEG or BMP), min-max rescaling is selected by default, and complex images are rejected when the format cannot store complex values
- Format-specific options are exposed and validated for each format (PNG compression level, JPEG quality/subsampling/progressive/optimization/smoothing, JPEG 2000 quality mode/layers/irreversible/progression/resolutions/tiling/PLT markers, TIFF compression/level/predictor/rows per strip/tiling/resolution/photometric interpretation, text and CSV delimiter and precision, MAT-File compression), and recorded History actions preserve the export parameters and target object UUIDs

Live processing preview:

- Parameter dialogs for compatible 1-to-1 processing now offer an optional live preview for signals and images, including image-to-signal operations. Preview is off by default; Cancel leaves the workspace unchanged, while OK reuses a completed current preview for a single source and otherwise performs the normal processing on the original selection.
- Completed previews keep their dedicated process ready for later dialogs, avoiding repeated process startup. Cancelling a preview that is still running stops its process so speculative work cannot continue in the background.
- Bounded numeric parameters now offer sliders alongside precise text entry. The Processing tab retains Apply and automatic re-processing, with updates deferred until slider release and valid input.

Plugin status:

- Click the Plugins: label in the status bar to open Configure plugins…, as with the Web API and internal console status labels. The shortcut is available whenever plugins are globally enabled, including when no plugins are loaded or a plugin fails to load.

Application plugins:

- Added a top-level Applications entry that shows the active scientific application plugins in a new Applications panel
- Each application displays its description, version, methods, tools, and examples; processing-only plugins remain outside this focused view
- Each method shows the inputs it expects (object type and count, required metadata) and a live status telling whether the current selection can be analyzed, and why not
- When required metadata are missing, the status points to Edit > Metadata > Add metadata…, whose Known keys list also offers the keys expected by application methods; the status is updated as soon as objects are modified

- Run… starts a method on the current selection directly when the selection is ready, and asks for the input assignment only when it is ambiguous or invalid; the assignment dialog checks the chosen objects before the run can continue
- Examples are listed under the methods they are designed for: Try with this example opens the example, prefills the method parameters, and runs the method once they are accepted. One example may serve several methods, and examples designed for no method are listed as datasets
- Application plugins may also list tools, such as wizards or editors, in the Applications panel
- Plugin tools also appear in the plugin's submenu of the Plugins menu, in the Signal or Image panel they work on. A tool that needs a selection is disabled until the right objects are selected, and its button in the Applications panel tells what to select
- Application plugins may provide instruments, such as simulated cameras or oscilloscopes, without writing any user interface: DataLab shows a window with a live view and the instrument settings, refreshed as settings change or continuously in Live mode, and Acquire adds the acquired objects to a new group, ready for the application's methods
- The New group option of the instrument window may be turned off to add the next acquisitions to the current group of the panel instead of creating a group for each of them. The option and the current group are taken when Acquire is clicked, so that selecting another group during an acquisition does not change its destination. Each instrument sets the default state of the option, and the choice of the user is remembered for each instrument
- Instruments are called outside the user interface thread, one call at a time: a slow instrument, such as a real device waiting for its data, no longer freezes DataLab, and repeated refresh requests made meanwhile are merged. Instruments may also declare `open` and `close` methods, called when their window opens and closes (including when plugins are reloaded or DataLab quits), for example to connect to a device and to release it. Calls stay one at a time when a window is closed and opened again while a call is running. When `open` fails, its message stays in the window and the instrument is opened again before the next refresh or acquisition, for example after a settings change
- Plugin developers describe recipe inputs declaratively (titles, minimum counts, metadata requirements) and may add binding suggestions and fast input checks; every recipe then runs through DataLab's generic launcher, without plugin-specific input dialogs
- The Applications panel is a regular panel that does not hide the DataLab workspace: the first time, it floats beside the main window, as high as this window, so that the data and their views stay in sight while using it. It may then be docked (for example next to the list of signals or images whose selection it analyzes), tabified with another panel or left floating, and DataLab shows it again where the user left it, with the same size; when it is requested while already shown, without switching to another application, it flashes briefly to show where it is. It remains open after starting a method or opening an example
- The panel stays compact and readable when docked: each method takes one row, with a colored dot and a short text telling whether the current selection can be analyzed, and a Run… button; below the list, a Details area gives the full details of the selected method (description, reasons of its status, expected inputs with the description and the key of their metadata, examples). The result of the last run of a method is shown below its row. The sections of the list may be collapsed, and the height of the details set with a splitter: DataLab remembers these choices
- When several application plugins are installed, their number is shown at the top of the panel, above the list selecting the application; the documentation and welcome page options of the application are in the menu next to it
- Completed application methods select their last generated object, consistently with standard DataLab processing
- Application plugins add tiles to a new Applications section of the welcome page: by default, one tile per plugin shows its page in the Applications panel, and plugins may declare additional tiles, for example to open an example directly. Additional tiles are shown when there is room for them, and otherwise move to the menu of the main tile
- The welcome page stays tidy when many applications are installed: tiles are limited to a configurable number of rows (two by default), with a last tile showing the Applications panel for the others. Applications may be pinned to the top or hidden, and recently used ones come first; the Applications panel offers the same welcome page options

- Plugins may declare an icon, shown in the Applications panel, on their welcome page tile and in the Configure plugins… dialog
- DataLab checks the declarations of application plugins (methods, examples, tools, welcome page tiles) when loading them: a plugin with an invalid declaration, for example a method whose plugin version was not updated with the plugin, or an example referring to a renamed method, is not loaded and is listed with its error in Configure plugins…, while DataLab and the other plugins start normally (fixes Issue #484)

Plugin installation:

- Online and offline Desktop catalogs now accept single-file `datalab_*.py` plugins alongside wheels. Historical files do not need a packaging project or source changes to be listed; catalog inspection never imports or executes them (implements Issue #419)
- Known script versions can be updated; unversioned files are identified by SHA-256 and offer explicit replacement when their content changes. Scripts previously installed from a file are recognized, and existing development copies are not overwritten
- The local catalog commands handle mixed folders of scripts and wheels. Single-file plugins use libraries already available to DataLab and do not trigger dependency downloads. The documentation now explicitly distinguishes source organization, distribution formats and functional capabilities, including in the French offline guide
- Plugins can now be installed from a file, without any Python tool, including in the standalone version of DataLab: the new From file tab of Plugins > Configure plugins… accepts a pure-Python wheel (`.whl`) or a single `datalab_<name>.py` module
- Before installing, DataLab checks the file without running it and shows its name, version, plugin classes, dependencies and SHA-256 digest
- Plugins may now include additional dependencies as separate wheels, including compatible native libraries, without changing packages already supplied by DataLab. A complete local catalog can be installed without Internet access; missing files and version conflicts are reported before any partial installation is activated
- The confirmation dialog lists the additional packages and their fingerprints. They are shared by plugins of the same Python runtime, activated after restarting DataLab, and kept when a plugin is uninstalled
- Installed plugins are listed in the same tab, where they can be uninstalled; a newly installed plugin is enabled after a plugin reload when no additional packages are needed, or after restarting when dependencies were added. A new version of an already loaded plugin is used at the next start
- The new Catalog tab lists the plugins of the DataLab plugin catalog, official or from the community, with a search field: Install and Update download the wheel, check it against the catalog and install it after confirmation. The catalog address may be changed to use a private copy
- Organizations can distribute their own plugins, or the versions of public plugins they have approved, from local plugin catalogs: folders, typically shared on the network, declared in the Advanced tab or with the `DATALAB_PLUGINS_CATALOG` environment variable. Their plugins are listed in the Catalog tab before those of the DataLab plugin catalog, and hide them when both offer the same plugin
- A local catalog folder may hold plugin wheels and single-file modules, listed as they are copied to it, or a `catalog.json` index giving display names, deprecations and withdrawn releases; `datalab-plugin catalog build` writes this index while keeping curated metadata, and `datalab-plugin catalog check` reports missing, changed, refused or unlisted artifacts
- Authors can prepare an offline catalog with `datalab-plugin catalog prepare`: it downloads the additional dependency wheels needed by the reference host without installing them, then writes the catalog. The complete folder can be distributed on a network share or copied to computers without Internet access
- A catalog that cannot be read, such as a network share that is not connected or the online catalog on a computer without Internet access, is reported in the Catalog tab without hiding the plugins of the other catalogs

- The Configure plugins… dialog opens larger and keeps the size it had when last closed. Its tabs follow the order of use and have explicit titles, described by their tooltips: Installed plugins, Catalog, From file and Advanced

Plugin project generator:

- `datalab-plugin create` now creates a single-file plugin by default, the simplest way to extend DataLab: it only asks for the plugin name and description, then writes a `datalab_<name>.py` file to the user plugin directory. Once the plugins are reloaded, the plugin adds its submenu to the Plugins menu with a sample processing of the selected signals (or images, with `--object-kind image`), whose short NumPy function is the only part to edit. The file never overwrites an existing one, and can be shared and installed with Plugins > Configure plugins… > From file. The layered installable project, described below, is created with `--project` (implements Issue #413)
- Hardened `datalab-plugin create` from the Camera pilot feedback: generated projects now separate host-independent `core` code, headless `workflow` orchestration, and Desktop/Web `adapters` from the first commit
- Generated package roots expose stable identity without importing Qt; the installed entry point targets the Desktop adapter while Web support is explicitly marked unsupported
- Every generated project includes an executable architecture regression test, contribution and architecture documentation, and a changelog
- Long plugin names and descriptions are now formatted so newly generated projects pass their bundled Ruff checks without manual source edits
- Generated projects are ready to publish: a release workflow attaches the wheel to a GitHub release when a version tag is pushed and prints the entry to submit to the DataLab plugin catalog; `pyproject.toml` requires the current DataLab major version and declares the project URLs
- `datalab-plugin create --project` asks for the GitHub account hosting the project and derives the default plugin ID from it (`io.github.<account>.<name>`), `org.datalab.` IDs being reserved for DataLab-Platform plugins

Add metadata:

- Edit > Metadata > Add metadata… can now extract the value from the formatted text with a regular expression, for example an exposure time or a shot number read from object titles. Objects without a match are left unchanged unless you ask for an error, and a scale factor converts numeric values (e.g. milliseconds to seconds)
- Metadata keys may now contain dots and hyphens, as used by plugin keys, and a Known keys list copies a key already present on the selected objects

Workspace provenance:

- DataLab now records how each result was obtained: the processing applied, its parameters, the objects it used and their exact content, and the software versions. This record is saved with the HDF5 workspace and read back when the workspace is reopened, by DataLab or by DataLab-Web, so the history of a result survives saving, closing and sharing
- Signal and image processing, operations combining several objects and analyses (geometry and table results) are recorded, each with the identity of its Sigima operation; regions of interest, uncertainties, the X alignment applied to signals on different grids and the global options read by the computation (such as kernel normalization) are part of the record, and signals and images of every numeric type (float32, complex…) are covered. An analysis that finds nothing is recorded as such. A result whose data, calibration or regions of interest were changed after it was computed is reported as altered rather than trusted
- Any recorded processing or analysis of a qualified Sigima operation (all of them but the phase spectrum) can be recomputed and every result compared with the stored one: exactly, or within the tolerance declared for the operation when the numerical libraries differ (exact, within tolerance, different, or not verified with the reason). Recomputation is available from the Provenance tab of the Properties panel (Verify computation) and from scripts; it never changes the workspace. When only the regions of interest of an input changed since the computation (for

example regions created on the analysed image by a detection), it is recomputed on a copy carrying the recorded regions of interest, so that such an analysis remains verifiable

- The History tab of the Properties panel is renamed Provenance (to avoid any confusion with the History panel) and now shows the provenance of the selected object: the recorded execution that produced its current content, its parameters and inputs, and, step by step, how each input was itself produced, as it was at the time (e.g. the sigma actually used, even if the input was filtered again since), with the software that recorded it (DataLab Desktop or DataLab Web). Each step is a card with its parameters, inputs and results, and links next to what they act on: show the recorded step of an input, Go to object, Verify this step, History action and details. Check data compares the object, its direct inputs and the earlier steps shown with their recorded states without computing; deleted inputs are reported at once, modified or unavailable ones after checking. Going to the History action of a step keeps the Provenance tab open
- The Processing tab restores the recorded call of the selected result, including operations combining several objects (difference of two signals, average of several images, …): every input is kept in order with its role, parameters can be edited, and processing without parameters offers a Reapply button. When the processing cannot be applied again (e.g. an input was deleted), the tab says why. This also works for workspaces reopened from a file, saved by DataLab or by DataLab-Web
- History actions are linked to the provenance of their last successful execution: Show execution provenance in the History panel displays it, and the provenance of a result leads to the History action that produced it, analyses included. In the History panel edit mode, editing one result of a command producing several results changes only its parameters in the action
- When a workspace file holding provenance is added to the current workspace (without clearing it), DataLab now says before anything changes that its earlier provenance will not be imported: in the question about clearing the workspace (files are chosen first), or in a single warning when that question is not asked. Added objects never inherit the provenance of another object (they are neither shown nor used as the original input of a recorded step, including when verifying it), and their later processing is recorded
- DataLab Desktop and DataLab-Web record the same processing identically and replay each other's results
- Before interpolating signals whose X grids differ, DataLab still asks for confirmation; the alignment it applies is the one it has always applied, now named and recorded so that a recomputation applies it identically. Several selected pairs of a two-signal operation now each align the operand on the grid of their own source signal
- Workspaces saved by earlier versions keep their processing history as stored: DataLab shows what is known and what is missing, invents nothing, and never recomputes such entries
- A workspace whose provenance record is damaged is refused with an explicit message, and the current workspace is left unchanged
- Workspaces can be exchanged as capsules (a workspace plus a description of its provenance) from File > Workspace capsule…, the same dialog as in DataLab-Web, so that someone else can check and replay them with either edition:
  - Export the workspace, with a name, a description, light attachments (protocol, notes, small tables) and, for a workspace opened from a capsule (also after it was saved and opened again), the capsule it is a revision of; the workspace can also be referenced by its publication address instead of being included
  - Inspect a capsule without opening it: its activities, its attachments (saved only on request, never opened), its integrity (workspace and stored states checked, nothing recomputed) and the verification reports and reviews supplied with it, each checked against the capsule; write a review of it, or Open as workspace (recorded in the History panel)
  - Local replay recomputes a recorded activity of the open workspace and compares it with its stored result, without changing the workspace; the verification report can be saved to be shared with the capsule
  - Revisions lists the capsules exported, inspected or retrieved on this computer and compares any two of them (scientific, presentation, storage, environment and attachment changes)

  - Registry connects to an optional capsule registry to publish capsules (never their workspaces) and reviews, list the published revisions and retrieve them, everything returned being checked
  - Scripts and macros can export and open capsules too (`export_capsule` and `open_capsule` of the proxy), and the `workspace-capsule` command line of DataLab-Capsule still validates, compares and verifies them
  - A capsule, or the workspace it references, that exceeds the size limit is refused before being read, and DataLab never writes a capsule it would refuse to open, nor writes the capsule or its referenced workspace over each other or over an attachment; the limits and the measured costs are given in the Workspace capsules page
  - The standalone version (installer) embeds what capsules need (schemas and RO-Crate context of DataLab-Capsule)

Object references in titles:

DataLab now uses stable UUID-based object references across object trees, result titles, plot legends and HDF5 workspaces (implements Issue #367 and Issue #149).

- Signal, image and group items in the trees now show their current title followed by their own stable `#UUID8` reference (`#gUUID8` for groups), replacing the former `s001`, `i001` and `gs001` identifiers
- Source references embedded in result titles no longer change when objects are reordered, removed or renumbered, and remain clickable to select the source object
- The new References in result titles setting lets you display these references either as 8-character UUIDs (default) or as the current source titles, which follow later renames and are resolved through whole processing chains (e.g. `derivative(derivative(My signal))`)
- When a source object is deleted, its reference falls back to its 8-character UUID instead of becoming an anonymous placeholder
- Titles of workspaces saved by previous versions keep their former `s001`-style references as plain text

Signal and image lists:

Signals, images and groups can now be deleted directly from the lists, as in DataLab-Web (implements Issue #449).

- A delete button appears at the right of the hovered signal, image or group: it turns red when the mouse cursor is over it, and its tooltip tells what it deletes
- Clicking it deletes the item, or the whole selection when the item is part of it, after confirmation. Deleting an item which is not selected keeps the current selection and the displayed objects unchanged
- As with the Remove action, the deletion is recorded in the History panel
- The button may be hidden with the new Show delete button when hovering list items setting (Visualization > Common)

Curve fitting formulas:

- Interactive fitting dialogs now show the formula of the fitted model below the fit parameters, and each parameter label shows its symbol in the formula (e.g. "Scale factor (x_0)"): the parameters stored in the fitted signal metadata can thus be related to the model without reading the source code. The Fitting documentation also gives, for each model, the name under which each parameter is stored (fixes Issue #400)
- Polynomial and linear fit coefficients are labelled `a`, `b`, `c`… as in the formula and in the metadata, instead of `c2`, `c1`, `c0`
- The Fitting documentation explains how to derive the peak wavelength, the temperature and the emissivity from the parameters of a Planckian fit

Remote control and plugins:

- The Contour isoline plot plugin (`org.datalab.isolevel-map`, version 1.1.0), formerly provided as a plugin example, is now a built-in plugin available directly from Plugins > Contour isoline plot > Show contour plot…

- `add_group` now returns the UUID of the created group, so that a remote client, a macro or a plugin can add objects to it with `add_object(obj, group_id=...)` (or `add_signal`/`add_image`) instead of looking the group up by its title
- `load_from_files` and `load_from_directory` now accept an optional `param` argument with format-specific import parameters (e.g. `RawImageImportParam` for RAW images), applied without showing any dialog

NeXus files:

- NeXus `NXdata` groups are shown as single items in the HDF5 browser and imported as signals or images with their axes, units, labels and, for signals, uncertainties. Bin edges are converted to bin centers, non-uniform image axes are kept, and decreasing image axes are reversed with the image. Invalid axes are reported in the browser instead of being guessed
- Files written with the legacy NeXus conventions, such as the scans of synchrotron SOLEIL, are imported with their axes (`axis` and `primary` attributes of the axis datasets), and every counter and detector of a scan is shown as an item of its own: counters as signals versus the scanned motor, detectors as images (a spectrum per point) or image stacks (an image per point) with the motor along their first axis
- The HDF5 browser selects and previews the default measurement of a NeXus file (`default` attributes) when it is opened
- Compressed detector data, such as the bitshuffle/LZ4 data of Eiger detectors, is now read (with hdf5plugin, also in the standalone application); a dataset compressed with an unavailable filter is reported in the browser instead of failing
- Imported signals and images record where they come from (file, group, signal and axes, applied conversions, title and times of the NeXus entry) in their `experimental_source` metadata, which is saved with the workspace: a saved workspace reopens without the NeXus file
- `.nxs` and `.nx5` files are proposed when opening HDF5 files
- Plugins can describe the signals and images of files that NeXus `NXdata` groups designate incompletely (no axes, several counters…) with declarative data mappings (JSON) registered with `sigimax.h5.mapping.register_mapping`: the HDF5 browser then shows each described object as an extra item, imported with its axes, labels and units; the axes of a mapped object may describe its first dimensions only, so that one mapping applies to the counters and detectors of a scan

Installation check:

- The new `--smoke-test` command line option checks that DataLab is correctly installed without opening its main window, e.g. `DataLab.exe --smoke-test report.json` for the standalone version or `datalab --smoke-test report.json`: the packages it relies on, the resources provided with it (macro templates, PDF documentation, French translations and icons) and processing in a separate process. The results are written in a JSON report, and the exit code is not zero when a check failed (see Command line features)
- Before being published, each Windows installer is now installed on a test computer, where the installed application is started and checked this way (implements Issue #366)

Changes

Curve readability:

- Signal curves are easier to read, especially when many curves are shown (closes Issue #453):
  - Curve colors now stand out on both light and dark backgrounds: several colors of the former palette (orange, pink, yellow-green, cyan) were pale on a white background, and gray curves could not be told apart from the grid
  - After the first 10 curves, curves are now drawn with dashed lines only: dash-dot and dash-dot-dot lines, which drew only half of the curve, are no longer used

  - The plot grid now uses discreet solid lines instead of gray dotted lines, which competed with dashed curves; it also follows color theme changes

- Curve colors and line styles now depend on the position of the curves among those shown on the plot, in the order of the signals in the panel: the first curve shown is always drawn in the first color. Opening the ROI editor, View in a new window or other dialogs no longer shifts the styles of the next curves, which could reach dotted styles with only a few curves shown
- Automatic curve styles are no longer saved in workspaces: only a color or a line style chosen by the user is saved and always kept, whatever its value. When opening workspaces and history files saved with DataLab 1.3 or earlier, which saved the automatic style of every curve, these styles are removed so that curves are displayed with the new automatic styles; a style chosen with these versions is removed as well when it is identical to one of their automatic styles. Metadata imported from a file are applied as they are
- View > Reset curve styles now gives the selected signals automatic curve styles again, removing the colors and line styles chosen for them
- On high-DPI screens (e.g. with a 150 % display scaling), curves of width 1 are now drawn one screen pixel wide, which makes their rendering much faster: noisy signals were drawn up to several hundred times slower than at 100 % (requires PlotPy ≥ 2.12.0)

Synthetic images in physical coordinates:

- The 2D Gaussian, 2D ramp, sinusoidal grating, ring pattern, Siemens star and 2D sinc images were computed over the [xmin, xmax] × [ymin, ymax] domain of their creation parameters, but were displayed in pixel coordinates: their centers, widths and spatial frequencies did not match the axes (a sinusoidal grating of frequency 0.1 over [0, 100] showed 10 periods along an axis graduated from 0 to 1023). These images are now displayed over their domain. The new Physical coordinates option of their creation parameters restores pixel coordinates when disabled (fixes Issue #400)
- Images created with earlier versions keep their pixel coordinates when their creation parameters are edited or recomputed

Guided tour:

- The guided tour no longer starts automatically at first launch: it is now offered from the welcome page, and remains available from the ? menu

Large campaign responsiveness:

- Opening, selecting, and displaying large generated campaigns no longer repeats a complete workspace and plot refresh for every object
- Native HDF5 workspaces and multi-object recipe results now load in groups, making large signal and image collections available substantially faster while preserving every selected and visible object

Image display responsiveness:

- Moving or resizing regions of interest, annotations and markers over images is now much smoother, as the image is no longer rendered again for each mouse move: with a 2048 × 2048 image, each mouse move takes about 5 ms instead of 13 ms (13 ms instead of 33 ms with a 200% display scaling). Panning and zooming images without masked pixels are also up to 25% faster (requires PlotPy ≥ 2.12.1)

Sharp icons with display scaling:

- The icons of the plots and of the editors provided by PlotPy and guidata (plot toolbars, contrast panel, cross sections, item list, array editor, …) are now SVG images: they remain sharp with a display scaling such as 125 %, 150 % or 200 %, instead of looking blurry (requires PlotPy ≥ 2.12.1, closes Issue #470)
- The Processing > Resize action of images showed the column resizing icon of guidata's editors instead of its own icon: DataLab icons now take precedence over guidata and PlotPy icons with the same name (requires guidata ≥ 3.17.1, closes Issue #472)

- Plugin icons (`PluginInfo.icon`, tools and welcome tiles) may also designate an icon registered by a module, e.g. `"datalab:delete.svg"`, in addition to `package:path` resources and DataLab icon file names

Portable plot annotations:

- Plot annotations are now stored in a renderer-independent format shared with DataLab-Web, so annotations in workspaces and `.dlabann` files are no longer tied to PlotPy
- Existing PlotPy annotations remain readable and are converted only after an annotation edit is accepted; simply opening a workspace or cancelling the editor leaves its data unchanged
- Annotation identifiers, lock state, custom metadata and extension data are preserved across edits, while unknown third-party payloads are retained without modification
- Rectangles and segments drawn on signals are now the same labeled shapes as on images; unlike on images, segments drawn on signals do not show their length and angle, which would mix X and Y units

Regions of interest:

- A rectangular region of interest now selects the pixels whose centers lie inside the rectangle, bounds included, like circular and polygonal ROIs and like the display: its bounds were rounded to the nearest pixel, which could shift the selection by up to half a pixel. A rectangle whose bounds lie exactly on pixel centers now includes the pixels on both bounds, i.e. one more row and one more column than before. This applies to processing and analysis in ROIs, to ROI extraction and to the background area of the offset correction
- Circular and polygonal regions of interest now also select the pixels whose centers lie inside their shape, boundary included: polygon vertices were rounded to the nearest pixel, so that the selected pixels could differ from the polygon drawn on the image, and circles were drawn in pixel units on images with non-uniform coordinates
- A rectangular ROI defined in pixel indices now covers its pixels up to their outer edges, so that it keeps the same pixels when converted to physical coordinates and back
- The cells of ROI grids (ROI > Create ROI grid…) are now made of whole pixels, so that adjacent cells never share a pixel and a full-size grid covers every pixel exactly once, whatever the image size; on images with non-uniform coordinates, the grid now spans the whole image, whose last rows and columns were left out
- Regions of interest saved with earlier versions are not converted: they select pixels according to these rules when they are used again, so that processing and analysis in these regions may give slightly different results when recomputed

Image profiles in physical coordinates:

- Line, segment, average and radial profiles (Analysis > Intensity profiles) are now defined in the physical coordinates of the image, like regions of interest and analysis results, instead of pixel indices: the profile parameters show coordinates in the units of the image axes, and the X axis of the resulting signals holds physical coordinates or distances with the matching unit. On images without calibration (origin 0, pixel size 1), results are unchanged
- The radial profile is now circular on images with rectangular pixels or non-uniform coordinates, and its center is correct on calibrated images, where the image center was misplaced
- Segment and radial profiles are refused on images whose X and Y units differ, since distances are then undefined; coordinates outside the image are refused instead of being silently moved to its edge. Lines and segments drawn beyond the image in the profile dialog are brought back to its outermost pixels
- Scripts, macros and plugins must use the new parameter names (`y`/`x` for line profiles, `x1`, `y1`, `x2`, `y2` for segment and average profiles, `xc`/`yc` for radial profiles) and physical coordinates. Profile computations recorded in the provenance of workspaces saved with earlier versions are reported as not verifiable, instead of being replayed with a different meaning

HDF5 browser:

- Browsing an HDF5 file no longer reads its datasets entirely: they are listed and described from their shape and data type, and the value previews read only their first elements. Large files open faster and use less memory; data is read when a signal or an image is previewed or imported
- The HDF5 browser stays responsive while a preview is read, and items selected meanwhile are not read; selecting an item that cannot be previewed (a group, for example) clears the preview. Ordinary 1D arrays and 2D images, NeXus and mapped objects above 2^{24} values (about 16.8 million) are previewed with a step along each axis (the axes of NeXus images are still read entirely, to check them as on import); compound datasets and signals stored in 2D tables still use complete objects for graphical previews
- Datasets with three dimensions or more (image stacks, spectral cubes…) are now listed in the HDF5 browser and imported as one image or signal per frame: a view selector chooses the displayed axes, the frames (first, last and step along another axis) and the index of the other axes, previews the first frame, and imports the frames once Import these frames is checked (with a confirmation beyond 100 frames). Frames of NeXus datasets keep their axes, and their title gives their position (e.g. the motor position of each detector image of a scan)
- Dataset attributes with more than 10 000 values are no longer copied to the metadata of imported objects

See the HDF5 browser guide for scientific object import, experimental origin and preview limits.

Scripts, macros and plugins:

- An object or a group looked up by its title, from a macro, a plugin, a remote client (e.g. `get_object("My signal")`) or the Web API, is no longer silently replaced by the first one found when several objects or groups share this title (e.g. files imported twice, "Untitled" objects): an error now names the matching objects with their UUID, which identifies them unambiguously (HTTP 409 for the Web API)
- For plugin developers: `patch_title_with_ids` no longer takes the third argument (`id_func`), and the helpers of the former `s001`-style identifiers (`SHORT_ID_REGEX`, `find_by_short_id`, `find_short_ids_in_title`, `replace_short_ids_by_uuids_in_titles`, `replace_uuids_by_short_ids_in_titles`) are removed from `datalab.objectmodel`, as titles now refer to objects by their UUID
- For developers of custom DataLab controllers derived from `BaseProxy`: the new `export_capsule` and `open_capsule` methods must be implemented

🛠 Bug Fixes

Performance with many objects:

- Fixed processing becoming progressively slower as signals or images accumulate in a panel: each new result rewrote the titles of every existing object, rebuilt the whole object tree and measured every title again, so creating 1000 signals one after the other took about 39 minutes; it now takes about 3 minutes (a processing step with 1000 objects in the panel dropped from 2.3 s to about 0.3 s)
- Image panels no longer slow down because of cross-section curves kept for hidden images (requires PlotPy ≥ 2.11.2)
- Fixed titles referring to objects numbered 1000 or above (e.g., `s1000`) being corrupted when objects were added or reordered

Window layout:

- Fixed the width of the signal/image panels not being restored at startup: after narrowing the panels and restarting DataLab, they came back wider (up to their maximum width) instead of keeping the width chosen by the user
- Fixed DataLab failing to start when its configuration file held an invalid window layout or an invalid log file name (e.g., after editing the file by hand): such settings are now ignored, and the default ones are used

Console:

- Fixed the console still being shown on errors after disabling Show console on error in the Settings dialog (with Qt 5, when the setting was enabled at startup), until DataLab was restarted

Demo:

- Fixed the demo failing with the error "No module named 'pytest'" in the standalone version of DataLab (installed with the MSI installer) and in installations without the test dependencies (e.g., `pip install datalab-platform`), whether started from the ? menu, the welcome page, the last step of the guided tour or the `datalab-demo` command (fixes Issue #365)
- The `datalab-demo` command no longer deletes the DataLab configuration of the user: the demo now runs with the default settings, without reading or changing the settings of the user

Standalone version:

- Fixed the standalone version of DataLab exiting with a success code when it failed to start: it now exits with an error code, so that scripts and deployment tools launching it can detect the failure (the cause is still written in `datalab_error.log`, in the current folder)

Display scaling:

- Fixed the splash screen, the DataLab logo of the welcome page and of the guided tour, and the watermark of empty plots being blurry with a Windows display scaling greater than 100% (e.g., 125%): they are now displayed at the resolution of the screen
- Fixed most of the text of the user interface (welcome page, object properties, panel tabs, dialog boxes, …) keeping its 100% size with a Windows display scaling greater than 100% (e.g., 125%), while menus and the status bar were enlarged. All the text now follows the display scaling and the Windows text size setting, and uses the standard font of Windows applications ("Segoe UI"). This affected DataLab running with Qt 5, like the standalone application (requires guidata ≥ 3.16.1)
- Fixed the line of the X axis of signal and image plots not being drawn with a display scaling such as 125% or 150%, depending on the size and position of the plot in the window (requires PythonQwt ≥ 0.17.1)
- Fixed images being displayed with a lower resolution than the screen with a display scaling greater than 100%: with a 200% scaling, images were shown with half the resolution of the screen, and with 125%, one screen pixel row and column out of five were duplicated, which made fine details (e.g., noise) look irregular. Images are now displayed at the resolution of the screen (requires PlotPy ≥ 2.12.1)
- Fixed the red plugin icon of the status bar (shown when some plugins failed to load) being too small and shifted to the top-left corner with a display scaling greater than 100% (e.g., half its size with 200%)

Dark theme:

- Fixed most windows keeping a white title bar with the dark theme on Windows, next to their dark content: the conversation history of the AI Assistant, the workspace capsule, the command palette and other dialog boxes, the import wizard, the confirmation and input boxes (e.g., when renaming a group or a conversation) and the panels detached from the main window (signal and image views, macro, history, AI Assistant and welcome panels, console). All the windows now have a dark title bar with the dark theme, also after changing the color mode in the Settings dialog (requires guidata ≥ 3.17.2, fixes Issue #490)

Plot watermark:

- Fixed the watermark of empty plots being read from another installation of DataLab sharing the same user configuration (e.g., DataLab installed in another Python environment): the watermark disappeared once this other installation was removed, and stayed blurry on high-DPI screens when it was an older version. The watermark of the running DataLab is now always used

Signal display:

- Fixed auto downsampling of signals (Settings > Visualization > Signals) only starting above 5 times the configured maximum number of points: with the default value of 100,000 points, signals of a few hundred thousand points were drawn in full, which slowed down the display. Signals with more points than the maximum are now downsampled so that at most this number of points is drawn, and the Settings dialog and the documentation now give the actual default value (100,000 instead of 10,000) (fixes Issue #455)

- Fixed changes of the auto downsampling and line width settings only applying to the curves of the selected signals: the other curves kept the former settings, even once selected again. These settings now apply to all curves as soon as the Settings dialog is closed
- Fixed the X axis ticks being shifted relative to the vertical grid lines and the curves after opening a signal or changing the selection, for signals whose X axis does not start and end on a graduation (e.g., `step.npy` from the Sigima test data): the shift remained until the view was panned or zoomed. The X axis now stays aligned with the grid and the curves (fixes Issue #457)
- Fixed curves of reloaded workspaces keeping the line width they had when the workspace was saved: changes of the line width setting (Settings > Visualization > Signals) no longer applied to them, and thick lines were no longer drawn 1 pixel wide for signals with many points, which slowed down their display. Only line widths chosen for a curve (e.g. in its parameters dialog) are now saved, and they are also drawn 1 pixel wide for signals with many points. When opening workspaces and history files saved with DataLab 1.3 or earlier, which saved the line width of every curve, line widths of 1 are removed so that curves follow the line width setting; other line widths are kept

Custom colormaps:

- Fixed custom colormaps (created with the colormap manager) being saved outside DataLab's configuration directory, in a `.plotpy` or `.none` directory of the user's configuration directory shared with other applications based on PlotPy, depending on how DataLab was started. They are now saved in the `plotpy` subdirectory of DataLab's configuration directory; the first time, custom colormaps saved in the former directories are copied there (requires PlotPy ≥ 2.12.0)

HDF5 import:

- Fixed images with unsigned 32-bit values above 2 147 483 647 being clipped when imported from an HDF5 file ("Clipping uint32 data to int32" warning): such images are now imported as 64-bit floating-point images keeping every value exactly. Other integer images that no supported integer type can hold also use a sufficiently precise floating-point type when possible, instead of rounding to 32-bit floating-point values beyond 16 777 216; a warning is emitted when no supported type guarantees exact conversion (in particular beyond $\pm 2^{53}$ for float64)
- Fixed single-precision complex (`complex64`) 1D datasets losing their imaginary part when imported from an HDF5 file: they are now imported as complex signals. Integer signals beyond $\pm 2^{53}$ now also emit the precision warning instead of being rounded silently
- Fixed the import of several datasets from HDF5 files stopping at the first dataset that could not be imported: the other datasets are now imported, and the datasets that could not be are listed in a single message at the end
- Fixed HDF5 files containing broken links failing to open in the HDF5 browser, for example synchrotron master files whose detector files are not available (external links to a missing file): the broken links are now skipped and the rest of the file can be browsed and imported
- Fixed HDF5 files containing a group linked into itself or into one of its subgroups failing to open in the HDF5 browser (the file was browsed endlessly until an error): such links back to a parent group are now skipped, while other aliases of a group or dataset are still shown at each of their paths
- Fixed datasets stored in other files, such as the detector data of synchrotron master files, missing from the HDF5 browser even when these files are available: the datasets reached through an external link were looked up at their path in their own file, so that they were skipped, or the data of another dataset of the opened file, found at that path, was imported instead. They are now listed and imported at their path in the opened file
- Fixed HDF5 files whose name contains a comma failing to open: the comma was taken for the separator between a file name and the name of a dataset to import. When appending such a workspace, the question about clearing the workspace also did not mention that its provenance is not imported, and an append cancelled by the user was still recorded in the History panel
- Fixed datasets holding a single row of values (e.g., a single scan stored as a 1 × N array) being silently skipped when imported from an HDF5 file: they are now imported and previewed as signals

- Fixed an error when clicking Show array in the HDF5 browser after selecting a group with the keyboard, or after closing the file of the selected dataset: the button now only shows arrays, and is disabled once the file is closed

Workspace saving:

- Fixed a failed save destroying the previous version of the workspace file: the file was emptied before the data was written, so that an error during the save, a full disk or an interruption left neither the previous nor the new workspace on disk. Workspaces, capsules and History files (`.dlhist`) are now written to a temporary file, which replaces the previous file only once completely written; if the save fails, the previous file is left untouched

File import:

- Fixed compressed TIFF files (LZW, JPEG or WebP compression) failing to open with the error "requires the 'imagecodecs' package"; LZW is the default compression of many acquisition and image-editing tools. These files now open out of the box, including with the standalone installer. CCITT-compressed bilevel images are also supported with the standalone installer and with Python 3.11 or later (fixes Issue #381)

Metadata:

- Adding, pasting or deleting metadata now marks the workspace as modified, so DataLab asks to save these changes before closing
- Importing metadata from a file into an object (Edit > Metadata > Import metadata) no longer gives it the identity of the object the metadata were exported from: two objects sharing an identity confused source references in titles, Select source objects and the saving of the workspace

Regions of interest:

- Fixed the Inverse ROI logic option of image ROIs being lost when validating the ROI editor: the option had no effect when set in the coordinates dialog of the editor (e.g., Rectangular ROI), and image ROIs created with it (e.g., by a macro) lost it once edited
- Fixed ROIs defined in pixel indices (e.g., by a macro or with the remote control) being converted to physical coordinates when the ROI editor was validated. They now stay in pixel indices, also when moved in the editor

Processing:

- Results of the extraction of several regions of interest into a single object can be processed again: the Processing tab showed "Unknown computing feature: extract_rois" with Reapply disabled, and replaying the History skipped the action
- Edit > Recompute now also recomputes results of operations combining several objects (e.g., difference of two signals, average of several images), which were skipped
- Fixed Processing > Resampling of images shifting every value by half a pixel: resampling an image with the default parameters, i.e. onto its own grid, filled its last row and column with NaN. The X/Y min/max parameters are now the coordinates of the centers of the first and last output pixels (by default, those of the image), and the 'Pixel size' mode keeps the requested pixel size exactly
- Fixed the sum, average, product and standard deviation of several signals, and their combination into an image, depending on the order of the signals: when a real signal came first, the imaginary parts of the complex signals were discarded
- The sum, average and product of several images now accept complex images, which raised an error
- Fixed Processing > Resampling of complex images with the default parameters, and the Z-axis Polynomial calibration of complex images, discarding the imaginary parts: complex images now stay complex
- Fixed modifying the result of a diagonal, horizontal or vertical flip or a 90° rotation of an image in place (e.g. from a macro or a plugin) also modifying its source image

- Fixed the extraction of several regions of interest into a single image giving wrong coordinates to images with non-uniform coordinates (the X coordinates were taken from the extracted rows and the Y coordinates from the extracted columns)
- The arbitrary rotation of images (Processing > Geometry > Rotate by…) now refuses images with non-uniform coordinates, like resizing and resampling, instead of producing an image whose coordinates did not match its pixels
- Fixed the uncertainty of the product of signals being lost where one of the signals is zero: it is now propagated there too
- Fixed ROIs, annotations and analysis results (centroids, detected peaks, contours…) being shifted by one pixel after a flip or a 90° rotation of an image; after a 90° rotation of a non-square image, or of an image whose coordinates do not start at zero, analysis results even ended up far from the features they describe. They now move exactly with the pixels; for images with non-uniform coordinates, circles, ellipses and polygons are moved approximately when the pixels are reversed along a non-uniform axis (a circle does not stay a circle). ROIs defined in pixel indices are also moved correctly, and inverse ROIs (which mask the inside of their shape) no longer become normal ROIs after these operations, a translation or a distribution of images on a grid
- Fixed ROI > Extract… giving the extracted image a wrong origin when the coordinates of the source image do not start at zero, and coordinates not matching the extracted pixels for images with non-uniform coordinates
- Fixed the coordinates of analysis results computed in a ROI being off by up to half a pixel
- Fixed the analysis results of images with non-uniform coordinates (Analysis > Centroid…, Minimum enclosing circle center…, 2D peak detection, Contour detection…, Circle Hough transform… and Blob detection) being given in pixel indices, or according to the former pixel size and origin of an image switched to non-uniform coordinates, so that they were drawn away from the features they describe: they are now placed at the positions of the pixels on the image axes, with radii scaled by the local pixel size. The results of images with uniform coordinates are unchanged
- ROI > Extract… now reports an explicit error for a region of interest holding no pixel of the image (for example a cell of a ROI grid narrower than a pixel, or a region outside the image) instead of silently creating an empty image; the other regions of interest are still extracted
- Fixed Processing > Level adjustment > Offset correction using a background area truncated to integer coordinates: the area was wrong for images whose coordinates are not in pixels (e.g. in millimeters), and empty when smaller than one unit

Plugins:

- Plugins organized as a folder, such as those created by `datalab-plugin create --project`, are now loaded when their folder is added to the plugin directories, without being installed first: they were silently ignored, unless an earlier installation had left package information in the project. The project folder itself can now be added, instead of its `src` subfolder (fixes Issue #411)
- Fixed small empty windows briefly flashing on screen when opening Plugins > Configure plugins…, just before the dialog appeared (fixes Issue #486)

Plugins and History:

- Replaying a History action or re-processing a result created by a plugin feature that shares its name with a built-in feature now runs the plugin feature instead of the built-in one

History Panel:

- Fixed an oversized warning appearing when deleting a group of signals or images whose processing results were kept in another group, even when the History panel was not recording: it listed every deleted object, often as an unreadable identifier, and stated that "operations could not be reconnected" although there was nothing to reconnect. Deleting such objects now shows no message, and the results depending on them are left unchanged as before. The warning is now shown only when a recorded processing step loses its source, and it lists each step once, in a message of bounded size (fixes Issue #451)

- Fixed the Description column of the History panel showing the raw content of voluminous parameters, e.g. the whole histogram of a Brightness and contrast action: the row grew to a huge height and scrolling the panel became erratic. The description now shows the actual parameters (e.g. the minimum and maximum of the brightness and contrast range) on a single line, shortened to the column width with the full summary in its tooltip; long values are shortened, and expanded details scroll inside a row of limited height (fixes Issue #369)

Object titles:

- Fixed plot legends still showing the original generated title after renaming an object, e.g. a profile extracted from an image (fixes Issue #323)
- Fixed source references in titles being unreadable when the object is selected in the signal or image tree, notably on Linux (fixes Issue #358)
- Fixed result titles pointing to the wrong source objects after appending an HDF5 workspace to a non-empty session; appending the same workspace twice now also keeps each imported processing chain independent, so Recompute and Select source objects use the sources of the same import (fixes Issue #357)

Curve fitting:

- Fixed the automatic Planckian fit (Processing > Fitting > Planckian fit) returning a curve about 150 times lower than the signal on data in physical units, such as a blackbody signal with wavelengths in meters; the interactive Planckian fit dialog also starts from a curve matching the signal peak (fixes Issue #400)
- Fixed exponential fitting failing on short time constants depending on the X unit, e.g. a 1 ms time constant with X in seconds (automatic and interactive fits)

Signal creation:

- The blackbody signal (Create > Blackbody (Planck's law)) now has axis titles and units (wavelength in m, spectral radiance in $W \cdot sr^{-1} \cdot m^{-3}$), which were missing (fixes Issue #400)

Remote control:

- Fixed files loaded remotely with `load_from_files` or `load_from_directory` being sometimes silently ignored, or still loading after the call had returned, when requested right after DataLab started: requests received while DataLab is starting are now processed as soon as its main window is ready (fixes Issue #398)
- Errors raised by remote commands, such as `calc` with an unknown computation function or `set_object` with an unknown object, are now reported to the client instead of being silently ignored
- Fixed remote commands sometimes returning the result of the previous command instead of their own: for example, `get_current_object_uuid` could return a panel name instead of `None` when there was no current object
- Fixed DataLab freezing when closed while a remote command was still running (e.g. when closing the main window with `call_method("close")`)
- Fixed DataLab crashing, or opening an error message box, when a request of a remote client or a macro failed, e.g. a computation requested with `calc` that failed on the selected signal, or a file that could not be read with `load_from_files`: the client was not informed of the error, and DataLab crashed when it ran without display (e.g. in the background, for tests or automation). The error is now reported to the client, as an `xmlrpc.client.Fault` with XML-RPC or as an HTTP error with the Web API (500 for a failed computation, 404 for an unknown computation function), and shown in the status bar of DataLab; this also applies to the datasets of an HDF5 file that could not be imported. Macros and remote clients now get an exception where they continued after the message box was closed (fixes Issue #404)
- Fixed every Web API request failing after 30 seconds with an HTTP 500 error ("Main thread execution timed out") when the Web API server had been started by a remote client with `start_webapi_server` (XML-RPC), including requests that should have failed at once, such as `calc` with an unknown computation function; the requested operation could even be carried out later, after the error had been returned. Requests are now served as when the server is started from the File > Web API menu (fixes Issue #403)

- Fixed images with non-uniform coordinates losing them when transferred through the Web API (`GET` and `PUT /objects/{name}/data`): they were rebuilt with default uniform coordinates. The NPZ data format now includes these coordinates; archives written by earlier versions remain readable. Clients with their own NPZ reader, such as DataLab-Kernel, must support the new entries to take advantage of this

AI Assistant:

- Fixed the Save to Macros link missing when a conversation is reopened from History…: a macro proposed by the assistant in a previous session could no longer be saved. Reopened conversations also showed every tool call as successful, including the failed ones; they now show the actual outcome (fixes Issue #447)
- Saving a macro proposed by the assistant now offers a Show in Macro panel link: the Macro panel is tabbed with the AI Assistant, so the saved macro stayed hidden behind the conversation
- Fixed the assistant answering a request to write a macro with the code in its reply, which could not be saved to the Macro panel: it now proposes the macro without running it (the confirmation dialog states that it will not be executed) and offers the Save to Macros link. A Save to Macros link is also offered under each Python code block of its replies, unless macro creation is disabled

Security:

- Opening a workspace received from someone else can no longer make DataLab load or run other code installed on the computer. The parameters stored with objects, analysis results and History actions name their class, and DataLab imported whatever module they named and called the object named there, which a modified file could use, for example, to make DataLab quit. Only parameter classes of DataLab, Sigima, guidata and installed plugins are now used; other stored parameters are reported as unreadable (fixes Issue #407)

5.2 Version 1.3

5.2.1 DataLab Version 1.3.0

✧ New Features

History Panel:

DataLab now records what you do and lets you replay it. A new History panel keeps track of every action applied to signals and images, from object creation and file loading to processing, analysis and region-of-interest edits (implements Issue #90).

- Recording sessions: actions are grouped into sessions, one active session per panel. Recording is disabled by default and is proposed when you create or load an object; configurable session policies control how plugin and macro objects are attached to sessions
- Processing chains: each session is presented as a linear processing chain, making the lineage of a result immediately readable
- Replay: replay a whole session, or step through it action by action, to reproduce a workflow on new data. Interactive curve fits and intensity profiles are recorded as deterministic actions and replay without reopening their dialogs
- In-place editing: processing, analysis and object-creation parameters can be edited directly from the history, and downstream steps are recomputed through the cascade
- Duplication: duplicate a chain to explore a variant without touching the original one
- Persistence: histories are stored in the HDF5 workspace and can also be exported and imported as standalone `.dlhist` files
- Compatibility checks: before replaying, DataLab validates the recorded actions against the current workspace state and reports what can or cannot be applied

- The Delete key removes the selected entries in the History panel (same behavior as in the Signal and Image panels), and only applies while the history tree has focus
- A resolution dialog is displayed when replaying a processing chain broken by deleted objects: choose "Repair and continue" to prune the broken steps and replay the remaining valid ones, or cancel without modifying anything
- A dedicated documentation page and a step-by-step tutorial are available

Built-in AI Assistant:

DataLab now ships an optional AI Assistant dock panel that can converse with you, inspect the workspace, create and process signals and images, and write and run Python macros (implements Issue #316).

- Works with any OpenAI-compatible endpoint: OpenAI, GitHub Models, Azure OpenAI, Ollama, LM Studio, etc. A mock provider is available to try the assistant offline, without any API key
- The assistant can call a set of built-in tools: list and inspect objects, introspect the available operations, create synthetic signals and images, load files, apply any registered processing, trigger plugin actions, and create and run macros
- Every action that modifies the workspace requires an explicit confirmation, with a parameter preview and a syntax-highlighted preview of the macro source when a macro is proposed. Read-only tools may be auto-approved
- Conversations are persisted, can be browsed and reloaded, and exported to Markdown; token usage and context size are displayed
- Settings (provider, model, API key, base URL, temperature, timeout, iteration cap, auto-approval) are grouped in a dedicated AI Assistant tab of the preferences dialog, with a connection test button
- The API key is stored in clear text in the DataLab configuration file, and the settings dialog states it explicitly

Command palette:

- A VSCode-style command palette lets you search and run any menu command by typing part of its name or of its menu path, e.g. "Processing ▸ Fourier analysis ▸ FFT" (implements Issue #337)
- Three entry points: a search box in the top-right corner of the menu bar, a "Command palette…" entry at the top of the Help menu, and the `Ctrl+Shift+P` keyboard shortcut
- Fuzzy matching ranks the results, so a few characters are enough to reach the right command, with full keyboard navigation

Spectral analysis support:

Following the spectral analysis work carried out in Sigima, DataLab gains the matching interface (implements Issue #309).

- New Analysis > Extract peak positions feature, storing the detected peaks as an XY-markers table result rendered as cross markers on the plot
- New Analysis > Peak detection… interactive dialog, to adjust threshold and minimum distance visually before storing the result
- New Operations > Create signal from markers table… feature, building a sticks signal from an XY-markers table result — convenient to overlay reference lines on a spectrum
- Marker results can be labelled in result tables (`#1`, `#2`, … for XY markers, `a`, `b`, `c`, … for axis markers), so each row can be matched with the corresponding marker drawn on the plot. This is controlled by a new "Show marker labels in result tables" option in the settings
- Result labels now support a customizable position and offset

Signal regions of interest rendered along the curve:

- Signal ROIs are no longer drawn as a single full-height yellow rectangle: the translucent fill is now clipped to the signal curve with a baseline at `y = 0`, and each ROI uses a different color taken from a cyclic ten-color palette (implements Issue #314)
- Colors are derived from the ROI index, so a given ROI keeps the same color across deletions, reorderings and editor reopenings
- Non-finite samples are filtered out before drawing, so signals with gaps render cleanly; on a logarithmic Y axis, the baseline falls back to the canvas bottom
- ROI persistence is unchanged: only coordinates are stored, visual properties are recomputed at render time

Regions of interest from detection (auto mask):

- Contour, blob, peak and Hough circle detection can create regions of interest directly from the detected features, turning intensity contours into editable polygon ROIs usable by every ROI-aware processing feature (implements Issue #294)

Third-party plugin discovery via environment variable:

- Added support for the `DATALAB_PLUGINS` environment variable, allowing one or more directories to be specified as additional plugin search paths (implements Issue #311)
- Multiple directories can be listed using the OS path separator (`;` on Windows, `:` on Linux/macOS), following the same convention as `PYTHONPATH`
- Listed directories are appended to the existing plugin search paths at startup and are picked up automatically by the plugin discovery mechanism
- Non-existent directories are silently skipped (a warning is recorded in the log file), so a stale environment variable on another machine will not prevent DataLab from starting

Plugin management user experience:

The plugin configuration dialog has been reworked to make third-party plugins easier to install, inspect and troubleshoot (implements Issue #302).

- Persistent additional plugin directories: several extra directories can be added, edited and removed from the dialog and are saved in the DataLab configuration, next to the read-only list of default directories
- Open file and Show in folder buttons per plugin, to inspect or edit a plugin script without hunting through the file system
- An **"Apply** and reload plugins" button and a Plugins > Reload plugins menu entry, to pick up plugin changes without restarting DataLab
- A last load timestamp displayed for the current session, to confirm that changes have been taken into account
- Plugin list filtering (all / enabled / disabled / with errors), a global enable/disable switch, and an option to hide compatibility warnings for legacy v0.20 plugins
- A Show in folder action is now available for the DataLab configuration file, from the installation and configuration viewer

Macro editor:

- Crash recovery: unsaved macros are auto-saved, and DataLab proposes to recover them at the next start
- Recent macros dialog, listing the macros used in previous sessions and allowing to reopen one in a single click
- Macro templates, to start from a working skeleton (simple macro, image processing, generic method call)
- Code completion and a find & replace widget in the macro editor

Object titles:

- Short identifiers embedded in object titles (e.g. `s001`, `i012`) are now rendered as clickable links in the signal and image trees, so the source objects of a computation can be selected in a single click

Replace special values processing (signal and image):

DataLab now provides a dedicated **"Replace** special values" processing function that detects and replaces `NaN`, `+Inf` and `-Inf` values in both signals and images. The feature is available under Processing → Level adjustment → Replace special values in both the Signal and Image panels (implements Issue #142).

- Each target (`NaN`, `+Inf`, `-Inf`) can be processed independently with its own strategy
- Signal strategies: do nothing, replace with zero / constant / minimum / maximum / mean / median, delete affected points, forward fill, backward fill, interpolation (linear, spline, quadratic, cubic, PCHIP), N-neighbor minimum / maximum / mean / median
- Image strategies: do nothing, replace with zero / constant / minimum / maximum / mean / median, N-neighbor minimum / maximum / mean / median
- The parameter dialog displays colored badges showing the number (and percentage) of `NaN`, `+Inf` and `-Inf` samples found in the source object, giving immediate visibility on what will be modified
- When a neighbor strategy is selected, a live kernel preview shows the shape of the neighborhood that will be used for the replacement
- Integer images are handled explicitly: because `NaN` and infinite values cannot exist in integer data, the dialog explains that the operation is not applicable and prevents accidental processing, while preserving the original image data type without unnecessary float conversion

↻ Changes

DataLab now builds on SigimaX:

- All the generic, application-level parts of DataLab (main window skeleton, configuration system, dockable plot widgets, HDF5 workspace and browser, log viewer, splash screen, status bar, scientific dialogs and PlotPy adapters) have been extracted into a new reusable library, SigimaX, and DataLab now derives from it instead of maintaining its own copies (implements Issue #182)
- This is an internal refactoring: existing workflows, settings and files are unchanged, but it considerably reduces duplicated code and makes it possible to build other Qt scientific applications on the same foundation
- As a consequence, DataLab now requires SigimaX ≥ 1.0.1, and its minimum requirements are aligned with it: Sigima ≥ 1.2.0, guidata ≥ 3.15.0 and PlotPy ≥ 2.11.0

Configuration system:

- DataLab's configuration has been migrated to the typed option system provided by SigimaX: options now carry an explicit type, default value and description, and are validated when read or written
- Settings recorded by earlier versions are migrated automatically at first start; the configuration file layout is otherwise preserved

`cdlclient` package archived:

- The standalone `cdlclient` package is deprecated and archived. Its lightweight XML-RPC client is now provided by Sigima, as `sigima.client.SimpleRemoteProxy` (implements Issue #183)
- Its Qt companion widgets have been taken over by SigimaX: the connection progress dialog is now `sigimax.widgets.connection.ConnectionDialog`, usable with any remote proxy exposing a blocking `connect()` method
- The documentation no longer refers to `cdlclient`

Detection tools now preserve existing regions of interest:

- Regions of interest created by peak, blob, Hough circle, and contour detection are now added to existing ROIs instead of replacing them, without displaying a destructive confirmation dialog (implements Issue #340)

Analysis results are now refreshed on demand instead of automatically:

- Analysis results (statistics, FWHM, centroid, peak/contour/blob detection, etc.) are no longer recomputed automatically when you modify a region of interest, transform the data, or edit object properties (implements Issue #341)
- Existing analysis results are now left untouched after such edits, avoiding surprising side effects and results that could become misleading once the data no longer matches the stored analysis parameters
- Ordinary replay and ordinary mutations do not recompute analyses; however, in History edit mode, editing upstream parameters recomputes downstream analysis actions through the cascade
- The familiar **"Recompute"** action (Edit menu, `Ctrl+R`) now refreshes both processing and analysis results, giving you full control over when analyses are updated
- For developers, the former panel-level `recompute_analysis` entry point has been renamed to `recompute_selected`; `recompute_analysis` now refers to a different processor-level helper dedicated to explicit 1-to-0 analysis refresh

🛠 Bug Fixes

Regions of interest:

- Fixed regions of interest created by a detection function (blob, contour, peak or Hough circle detection) reappearing immediately after being deleted or modified — detection ROIs are now ordinary, fully editable ROIs (fixes Issue #329)
- Fixed deleted regions of interest reappearing after a post-processing automatic recompute
- Fixed a crash when the region-of-interest index was out of bounds while building an ROI title

Intensity profiles:

- Fixed the "Reset selection" button of the intensity profile dialogs not clearing the previous selection: after resetting and drawing a new region, the extracted profile still corresponded to the first selection (fixes Issue #322)
- Fixed profile recompute using the parameters of another object when several objects were selected

Command palette:

- Fixed search results being irrelevant for short queries — typing "rota" no longer returns unrelated commands before the rotation entries (fixes Issue #345)

Full width at half maximum:

- A user-reported FWHM regression observed in DataLab v0.20.1 (an aberrant, far too small width) has been pinned down by a non-regression test built on the original signal, now part of Sigima's test data. The computation was already correct in v1.2.x and remains correct in this version (fixes Issue #356)

Plugins:

- Fixed plugin processing functions failing in spawned worker processes after a plugin reload

Main window:

- Fixed the visible plot dock not following the current panel tab in some situations, in particular when a persisted layout left the Macro, History or AI Assistant dock raised at startup

File saving:

- Fixed an error raised when saving a signal to a file in some situations

History Panel:

- Fixed replay of load actions doing nothing when the loaded objects had been deleted while recording was active — the objects are now reloaded from their files
- Fixed duplicated processing chains interfering with their source chain: replaying either session now only affects its own objects
- Fixed session replay aborting when the session contained a recorded object deletion — recorded deletions that can no longer be safely applied (captured state mismatch, targets re-created earlier in the same replay, or targets belonging to another session) are now skipped with a warning
- Removed the yellow highlight that could persist on history entries after a failed replay
- Fixed outdated steps having no visual indication in the history tree — actions left outdated by a failed or interrupted recompute, or by parameter edits pending propagation, are now highlighted with an amber background and a tooltip suggesting to replay them
- Fixed potential crashes or corrupted histories when clicking History panel commands (Replay, Delete, Duplicate, …) while a long replay or recompute was still running — commands are now unavailable until the run completes
- Fixed a single corrupted history entry (e.g. a ROI saved by an incompatible version) preventing an entire `.dlhist` file or HDF5 workspace history from loading — the affected action now loads as incompatible while the rest of the history loads normally
- Fixed recording aborting the user's operation when a selected object had no data
- Fixed the active session remaining highlighted after loading an HDF5 file
- Fixed deleting an image history action switching to the Signal panel
- Fixed a macro generation error (`object has no attribute uuid`) for 2-to-1 processing pipelines
- Improved performance: opening many files or replaying long processing chains no longer freezes the interface while the History panel repeatedly rechecks action compatibility

📖 Documentation

- Added a complete History Panel feature page and a step-by-step tutorial covering recording, replay, parameter editing and session import/export
- Added use case pages illustrating DataLab in photonics, spectroscopy and non-destructive testing
- Documented the legacy HDF5 dataset selector syntax
- Reworked the documentation layout: more compact navigation bar, Open Graph metadata, and software citation metadata
- Added a plugin example showing iso-level contour lines on an image
- Updated French translations across all new and modified documentation pages

5.3 Version 1.2

5.3.1 DataLab Version 1.2.2

🛠 Bug Fixes since version 1.2.1

Signal peak creation and fitting:

- Gaussian, Lorentzian and Voigt creation and fit dialogs now express amplitude as signed peak height above the baseline, in the signal's Y units. Objects carrying historical area-based peak parameters remain displayable and keep their original samples; editing or re-evaluating them now requires an explicit conversion confirmed by the user (fixes Issue #350)
- Interactive multi-Gaussian and multi-Lorentzian results now retain every detected peak position (`x0_i`) in canonical fit metadata, allowing complete theoretical models to be exported or re-evaluated on another X axis without adding center controls to the fit dialog (fixes Issue #351)
- The "Evaluate fit" feature is now available for every interactive fit, not just peak fits: linear, polynomial, exponential, sinusoidal, CDF, Planckian, two half-Gaussian and piecewise exponential fitting curves may now be re-evaluated on another X axis or exported as theoretical models (fixes Issue #355). Fitting curves computed by earlier versions carry parameters that were labelled in the interface language and cannot be re-evaluated: DataLab now reports this explicitly and such curves have to be recomputed.
- Scripts and remote clients may migrate historical fit metadata explicitly by retrieving an object with `get_object()`, converting a copy with Sigima's `convert_legacy_peak_fit_params()`, and updating the existing object with `set_object()`; the conversion never regenerates the signal samples.
- Fixed interactive fit parameter sliders whose bounds excluded reachable solutions: the exponential `B` slider could not express a decay, the cumulative distribution function amplitude could not express a descending transition, and the piecewise exponential amplitude bounds were inverted for negative initial guesses (fixes Issue #352)
- Fixed the Planckian fit dialog labelling the `x0` parameter as the peak wavelength — the maximum of the model is located at approximately `1.007 * x0 / sigma`, so the two only coincide when `sigma` equals 1

HDF5 import:

- Fixed a crash when running a computation on an object imported from an HDF5 file containing object or region reference attributes (e.g. `DIMENSION_LIST` / `REFERENCE_LIST` arrays produced by `h4toh5convert`) — such attributes cannot be serialized to worker processes and are now skipped instead of being copied to the object metadata (fixes Issue #346)

Text import wizard:

- Fixed a sporadic application crash when closing the Text Import Wizard — the embedded plot preview is now released deterministically instead of being torn down later by the garbage collector (fixes Issue #335)

Working directory handling:

- DataLab no longer changes its process working directory when a file is opened or saved. The last used folder is still remembered for file dialogs, but the directory itself is no longer locked for the lifetime of the application on Windows (it can be moved, renamed or deleted while DataLab is running), and macros or scripts started from DataLab no longer inherit an unexpected working directory

Packaging:

- Fixed missing data files in the distributed package: all `.h5` test/demo files and `.template` files are now included in the wheel and source distribution

📖 Documentation since version 1.2.1

- Added a new "DataLab Platform ecosystem" page positioning DataLab (desktop), DataLab-Web (browser edition), DataLab-Kernel (Jupyter kernel) and Sigima (computation engine) with respect to each other; the browser edition is now surfaced on the home and getting started pages, plugin web compatibility is documented, and the roadmap was refreshed (web frontend and Jupyter kernel delivered)
- Added NixOS installation instructions: DataLab is packaged for NixOS and distributed through the NGI Forge, thanks to a contribution from the Nix@NGI team as part of DataLab's funding through the NGI0 Commons Fund
- Added a "Talks & Events" section to the documentation, starting with the EuroSciPy 2026 presentation details
- Fixed the polynomial signal creation formula, which listed a `y0` term and an arbitrary number of coefficients whereas the parameters actually expose `a0` to `a5`, and documented the sigmoid fit, which was available but undocumented (fixes Issue #353)
- Aligned the documented signal model formulas with the new height-based parameterization of the Gaussian, Lorentzian and Voigt models
- Refreshed the signal and image "Edit" documentation examples, which still showed object titles produced by older versions (fixes Issue #354)
- Visiting the documentation site root now redirects to the French or English version depending on the browser language, instead of always serving the English pages
- Updated French translations across all new and modified documentation pages

🔧 Improvements since version 1.2.1

Compatibility:

- Updated minimum Sigima requirement from 1.1.3 to 1.1.6 (computation fixes and versioned peak-height parameters introduced in 1.1.6)

5.3.2 DataLab Version 1.2.1 (2026-06-05)

🛠 Bug Fixes since version 1.2.0

Signal panel menus:

- Restored the "Linear calibration" entry in the "Processing > Axis transformation" menu of the Signal panel — it had been inadvertently dropped during a menu reorganization and was no longer reachable from the menu bar, even though the underlying computation was still available (fixes Issue #312)

Signal panel — Y range cursor:

- Fixed the Y range cursor annotation displaying an incorrect inequality (e.g. `5 < y < 2`) and a negative Δy when the top cursor was dragged below the bottom cursor — the annotation now always shows values in sorted order with a positive range width (fixes Issue #306)

Image panel — Z-axis logarithmic scale:

- Fixed the "Base-10 logarithmic Z axis" toolbar action being permanently greyed out in the image panel — the tool now works correctly for all image types (fixes Issue #313, fixed upstream in PlotPy ≥ 2.10.0)

Image panel — contour detection:

- Fixed ellipse and circle contour detection producing incorrect results (wrong positions and sizes) due to swapped X/Y coordinates and missing unit conversions in the scikit-image model fitting code (fixes Issue #326, fixed upstream in Sigima ≥ 1.1.3)

ROI editor:

- Fixed the ROI editor dialog ignoring non-linear axis scales (e.g. logarithmic) of the source plot — the dialog now preserves the same axis scale configuration as the main panel (fixes Issue #315)

Theme and display:

- Fixed plot marker and shape colors being reverted to PlotPy defaults (yellow) instead of DataLab's configured colors (red) when the color mode was explicitly set to "light" or "dark" in settings — the custom color overrides are now re-applied after each theme switch (fixes Issue #297)
- Fixed UI elements (text, icons, dialog layouts) not scaling properly with high-DPI display settings — DataSet HTML rendering, icon sizes, and viewport dimensions now adapt to the system scale factor (fixes Issue #317, fixed upstream in guidata ≥ 3.14.4)

Image properties editor:

- Fixed pixel size (`Δx` / `Δy`) input field corrupting typed values — most digits typed after the first were misinterpreted as decimal places due to a reactive update loop in the parameter editing widget (fixes Issue #320, fixed upstream in guidata ≥ 3.14.4)

Debug environment variable renamed:

- Renamed the debug environment variable from `DEBUG` to `DATALAB_DEBUG` — the generic `DEBUG` name collided with widely-used third-party conventions (Django, Flask, Node.js tooling, CI systems) and could silently reset the user configuration file at startup when set for unrelated reasons. Setting `DATALAB_DEBUG=1` now activates debug mode; the bare `DEBUG` variable is ignored (fixes Issue #319)

Remote control API:

- Exposed `get_current_object_uuid()` on the proxy API, making it available through XML-RPC and Web API

🔧 Improvements since version 1.2.0

Compatibility:

- Updated minimum guidata requirement from 3.14.3 to 3.14.4 (high DPI and screen scaling issue, dataset input field fix, add secure build CLI cmd)
- Updated minimum PlotPy requirement from 2.8.2 to 2.10.0 (Z-axis log scale fix, toolbar overflow button visibility in dark theme, PythonQwt 0.16.0 optimization)
- Updated minimum Sigima requirement from 1.1.2 to 1.1.3 (ellipse and circle contour detection fix)

5.3.3 DataLab Version 1.2.0 (2026-04-20)

✨ New Features since version 1.1

Plugin configuration dialog:

DataLab now provides a dedicated plugin configuration dialog (accessible via "Plugins > Configure plugins…") that gives full control over third-party plugin management:

- Enable or disable individual plugins using checkboxes, or toggle all plugins at once with a tri-state master checkbox
- Filter plugins by status: all, enabled, disabled, or plugins with import errors
- View plugin details including version, author, and expandable long descriptions directly in the dialog
- Plugins with import errors are displayed prominently at the top with their full traceback, making it easy to diagnose installation issues
- The expandable text widget used for long descriptions computes its preferred width from a fixed measurement context, ensuring stable layout and reliable "Show full description" toggling regardless of dialog resizing or offscreen rendering

Plugin hot-reload:

- Third-party plugins can now be reloaded at runtime without restarting DataLab, via "Plugins > Reload plugins"
- Enabling or disabling plugins in the Preferences dialog or plugin configuration dialog takes effect immediately - no restart required
- The Plugins menu, status widget, and plugin actions are automatically refreshed after configuration changes

Multi-instance detection:

DataLab now detects when another instance is already running and warns the user before opening a second instance:

- Uses a PID-based lock file mechanism that supports multiple concurrent instances (reference counting)
- Stale PIDs from crashed processes are automatically cleaned up
- Cross-platform support (Windows, macOS, Linux) using platform-specific process detection
- Closing one instance no longer incorrectly removes the lock file when other instances are still running

Image ROI editor contrast synchronization:

- The image ROI editor now shares contrast (LUT) settings with the source image panel
- Adjusting the contrast in the ROI editor is reflected back in the main panel and vice versa
- Contrast controls are fully re-enabled in the image ROI editor dialog

Remote control API — push modified objects back to DataLab:

The proxy API (XML-RPC and Web API) now exposes a new `set_object` method that updates an existing signal or image in DataLab from a modified copy obtained via `get_object` (fixes Issue #305):

- Previously, modifications to object properties (e.g. `dx`, `dy`, `x0`, `y0`, `title`) made on the result of `get_object` were lost because `get_object` returns a copy — `set_object` now provides a clean round-trip workflow
- Works for both signal and image objects: computed result items attached to the object are preserved during the update, so updating an `ImageObj` no longer triggers a type mismatch
- The properties panel is automatically refreshed after `set_object`, so updated object properties (title, units, axes, uncertainties, etc.) are immediately visible in the GUI

🕮 Documentation since version 1.1

- Added API documentation for the `datalab.objectmodel` module
- Added screenshots for the "Paste metadata" dialog (signal and image panels)
- Updated plugin documentation to describe the new configuration dialog, hot-reload workflow, and plugin API helpers
- Updated third-party plugin development guide with new template references and test coverage information
- Expanded Web API reference: documented the binary data transfer options (`?compress=false` for faster uncompressed NPZ downloads, `?overwrite=true` for atomic replacement of existing objects), the in-place `PUT /objects/{name}` endpoint that updates an object while preserving its identity, group membership and position, and the new "Computation" section listing the `/select` and `/calc` endpoints used to drive DataLab computations remotely
- Updated French translations across all new and modified documentation pages

🔧 Improvements since version 1.1

Compatibility:

- Officially support pandas 3.0.x (updated dependency constraint from `< 3.0` to `< 3.1`)
- Updated minimum Sigima requirement from 1.1.0 to 1.1.2 to benefit from latest computation engine fixes and improvements
- Added legacy support for the `WINPYDIRBASE` environment variable for WinPython-based deployments

Plugin system hardening:

- Plugin discovery and registration is now resilient to third-party import failures while preserving error reporting in the console, logs, and configuration dialog
- Plugin submenus are now scrollable to prevent overflow when many plugin entries are registered

Development tooling:

- New `run_with_env.py` script for running tasks across multiple Python environment contexts (WinPython, venv, etc.)
- Simplified environment variable handling by removing the `DATALAB_ENV_LOADED` system
- Fixed Coverage full VS Code task to properly use the `run_with_env.py` wrapper

🛠 Bug Fixes since version 1.1

HDF5 workspace - Table serialization:

- Fixed callable metadata not being stripped during HDF5 save/load of `TableResult` objects
- Fixed string-based enum values in table results not being restored correctly after HDF5 round-trip
- Fixed `column_formats` attribute not surviving HDF5 round-trip for both `TableResult` and `TableResultBuilder` outputs

Plugin system:

- Fixed `AttributeError` in plugin configuration dialog when clicking "Show full description" (incorrect attribute reference)
- Fixed plugin import errors being silently swallowed when they occurred before the internal console was initialized

5.4 Version 1.1

5.4.1 DataLab Version 1.1.1 (2026-04-08)

> Note: This is a maintenance release that fixes a single bug affecting only the stand-alone executable version of DataLab for Microsoft Windows. If you installed DataLab via `pip`, `conda`, or if you are running DataLab on Linux or macOS, you are not affected by this bug and there is no need to update to this version.

🛠 Bug Fixes since version 1.1.0

Flip/rotation features crash or hang in stand-alone version:

- Fixed image flip and rotation features ("Flip horizontally", "Flip vertically", "Rotate 90° left/right", "Flip diagonally") crashing with an `ImportError` or hanging indefinitely in the stand-alone Windows version (PyInstaller build)
- The issue was caused by the multiprocessing worker process needing to import the full DataLab GUI module chain when unpickling the geometric transform wrapper, triggering circular imports in the frozen executable

- Moved the geometry post-processing code to a standalone module with no GUI dependencies, so the worker process no longer triggers the DataLab GUI import chain
- This was not reproducible with the Python package version of DataLab, only with the stand-alone executable
- This closes Issue #304 - Flip/rotation features crash or hang in stand-alone (PyInstaller) version

5.4.2 DataLab Version 1.1.0 (2026-02-02)

✧ New Features

Web API for HTTP/JSON integration:

DataLab now provides a modern HTTP/JSON Web API as an alternative to the existing XML-RPC interface. This is the recommended integration path for new projects, especially for Jupyter notebooks and web-based tools.

- Accessible via: File → Web API → Start Web API Server (or set `DATALAB_WEBAPI_ENABLED=1` environment variable)
- Key benefits:
  - JSON for metadata and NPZ for binary data (efficient large array transfer)
  - Bearer token authentication for security
  - REST conventions with OpenAPI documentation
  - WASM/Pyodide compatibility (works in browser environments)
- Endpoints include:
  - Object listing, retrieval, creation, and deletion
  - Metadata management and binary data transfer (NPZ format)
  - Computation API for running processing operations remotely
- Auto-discovery: DataLab-Kernel automatically finds and connects to a running DataLab instance without manual configuration
- Security: Localhost binding by default, with optional localhost token bypass for simplified local development

PyQt6 compatibility:

- DataLab now fully supports PyQt6 in addition to PyQt5
- Fixed screen geometry retrieval to use `primaryScreen()` for Qt5-Qt6 compatibility
- CI pipeline now includes PyQt6 testing to ensure ongoing compatibility

Remote control API enhancements:

> Note: These new features also concern the macro commands API, as macros use the same proxy interface as remote control clients.

- Added `call_method()` method to `RemoteProxy` class for calling any public method on DataLab main window or panels:
  - Enables programmatic access to operations not exposed through dedicated proxy methods (e.g., `remove_object`, `delete_all_objects`, `get_current_panel`)
  - Supports automatic method resolution: checks main window first, then current panel if method not found
  - Optional `panel` parameter allows targeting specific panel ("signal" or "image")
  - Thread-safe execution: GUI operations automatically execute in main thread to prevent freezing

  - Proper exception handling: exceptions raised during method execution (e.g., attempting to call private methods) are captured and propagated to the client as `xmlrpc.client.Fault` objects with the original error message
  - New macro example `test_call_method.py` demonstrates usage: removing objects, getting current panel, and panel-specific operations
  - Expands automation capabilities for advanced macro workflows and external application control
- Added `remove_object()` method to proxy interface (local and remote) for selective object deletion
  - Removes currently selected object from active panel
  - Optional `force` parameter to skip confirmation dialog
  - Complements existing `reset_all()` method which clears the entire workspace (including all macros)

Dependencies:

- Updated Sigima dependency to version 1.1.0 which includes new features and bug fixes
- Added Web API dependencies: FastAPI, Uvicorn, and Pydantic (included in standard installation)

5.5 Version 1.0

5.5.1 DataLab Version 1.0.4 (2026-02-02)

🛠 Bug Fixes since version 1.0.3

Centroid computation compatibility:

- Fixed centroid computation failing with NumPy 2.4.0 and scikit-image 0.26.0 due to new einsum optimization that doesn't accept masked arrays
- Image data is now explicitly converted to a standard NumPy array before computing the centroid
- This closes Issue #289 - Compatibility issue with NumPy 2.4.0 and scikit-image 0.26.0

5.5.2 DataLab Version 1.0.3 (2025-12-19)

🛠 Bug Fixes since version 1.0.2

Loading images with a single row or column fails:

- Fixed loading image files with a single row or column (e.g., some SIF spectroscopy files with shape `(1, N)`) causing an `IndexError` and preventing the image from being displayed
- The underlying issue was in PlotPy's coordinate-to-bin conversion function which assumed at least 2 points
- This fix is included in PlotPy 2.8.3

LUT range incorrectly copied when processing images:

- Fixed processed images inheriting the LUT range (display contrast settings) from the source image, causing incorrect visualization when data values change significantly
- When applying operations like offset correction, the result image now auto-scales its display range to fit the new data values instead of using the original image's LUT range
- Previously, users had to manually adjust the LUT to see processed images correctly (they could appear completely black or with wrong contrast)
- This closes Issue #288 - LUT range incorrectly copied when processing images

- This fix is included in Sigima 1.0.4

Separate plot windows - Incorrect aspect ratio for non-uniform coordinates:

- Fixed separate plot windows (e.g., "View in a new window", ROI editors, profile dialogs) not using the same aspect ratio configuration as the integrated plot handler
- When displaying images with non-uniform coordinates, the main plot correctly unlocks the aspect ratio for proper display, but separate windows were not inheriting this setting
- All image dialogs now properly inherit the aspect ratio settings, ensuring consistent display behavior across all plot windows
- This closes Issue #287 - Separate plot windows don't use same aspect ratio configuration as integrated plot

Remove all results - AttributeError when ROI was removed:

- Fixed "Remove all results" action failing with `AttributeError: 'NoneType' object has no attribute 'get_single_roi_title'` when results contain ROI information but the ROI was subsequently removed from the object
- The issue occurred when:
  1. Running an analysis with a ROI (e.g., Centroid on an image with rectangular ROI)
  2. Removing the ROI from the object
  3. Trying to delete all results via "Analysis > Remove results > Remove all results"
- The fix adds a proper None check before accessing the ROI's title
- This closes Issue #286 - "Remove all results" fails with AttributeError when ROI was removed

Analysis auto-recompute - Stale parameters after deleting results:

- Fixed analysis parameters not being cleared when deleting analysis results, which could cause unexpected auto-recompute behavior when modifying ROIs
- After deleting results (via "Delete all results" or individual result deletion), changing the ROI would trigger recomputation of the deleted analysis because the stored parameters remained in object metadata
- The fix ensures analysis parameters are properly cleared alongside the results, preventing unwanted automatic recomputation
- This closes Issue #285 - Analysis parameters not cleared after deleting results

Backwards-drawn rectangular ROI causes NaN statistics:

- Fixed rectangular ROI statistics returning NaN values when the ROI was drawn "backwards" (from bottom-right to top-left instead of top-left to bottom-right)
- The issue occurred because the ROI coordinate conversion produced negative width/height values, causing the mask generation to fail
- ROI mask generation now works correctly regardless of the direction in which the rectangle was drawn
- This closes Issue #284 - Backwards-drawn rectangular ROI causes NaN statistics
- This fix is included in Sigima 1.0.4

Plot refresh - ROIs and annotations persist after "Remove all":

- Fixed ROIs and annotations occasionally remaining visible after executing "Remove all" action from the Edit menu
- The issue occurred because the plot widget wasn't explicitly told to redraw after clearing all plot items and shape items

- The plot now correctly refreshes its display after clearing all objects, ensuring a clean view without requiring panel switching
- The bug was intermittent and hard to reproduce systematically, making it appear as if objects were "stuck" on the plot

Result labels - False "omitted" message for single-row results:

- Fixed result labels (e.g., pulse features) incorrectly showing "X more rows omitted" message when displaying single-row results with no actual truncation
- The bug occurred because `len(adapter.result)` was used to get the row count, but this actually returns the number of column headers, not data rows
- For pulse features with 13 columns, this caused the label to think there were 13 rows when only 1 row existed, leading to a false "12 more rows omitted" message
- The fix ensures accurate row counting using `len(df)` instead, preventing false truncation notices
- This closes Issue #281 - Result Labels False "Omitted" Message

Signal view - XRangeSelection items persist after removing signals:

- Fixed XRangeSelection and DataInfoLabel items (created by PlotPy's curve statistics tools) remaining visible on the plot after removing all signals
- When activating the XRangeSelection tool from the PlotPy toolbar and then removing the signal(s), these tool-created graphical objects were not cleaned up
- The cleanup logic now correctly triggers when there are zero selected objects (after deleting the last one), not just when exactly one object is selected
- This closes Issue #280 - XRangeSelection items persist after removing all signals

Remote control - Crash when adding object via proxy:

- Fixed `AttributeError: 'NoneType' object has no attribute 'setText'` crash when adding objects via `RemoteProxy.add_object()` in macros
- The crash occurred in `update_tree()` when the tree view became temporarily out of sync with the object model (e.g., when adding objects with specific metadata configurations or ROIs via remote proxy)
- The tree view now rebuilds automatically if inconsistencies are detected, ensuring robust object addition in all scenarios
- This closes Issue #279 - AttributeError in update_tree() when adding object via RemoteProxy

Icons - Qt SVG warnings on Linux:

- Fixed "qt.svg: Invalid path data; path truncated" warnings appearing 4 times at startup on Linux/Ubuntu
- Cleaned SVG icon files to remove Inkscape/Sodipodi metadata and fixed path syntax in two icon files that used implicit cubic Bézier command continuation (not supported by Qt's SVG renderer on Linux)
- This closes Issue #278 - Fix "qt.svg: Invalid path data; path truncated" warnings on Linux

ROI extraction - KeyError when extracting ROIs:

- Fixed `KeyError` when extracting ROIs from images with ROIs generated by blob detection or other analysis functions
- The issue occurred because the ROI editor was trying to access plot items that hadn't been created yet (e.g., when "auto refresh" was disabled or in certain selection scenarios)
- The ROI extraction dialog now handles missing plot items gracefully
- This closes Issue #276 - KeyError when extracting ROIs from images with blob detection results

Plugin development - Infinite loop when iterating over group objects:

- Fixed infinite loop occurring in plugins when iterating over object IDs returned by `ObjectGroup.get_object_ids()` while simultaneously adding new objects to the same group
- The method was returning a direct reference to the internal list instead of a copy, causing the iteration to include newly added objects and loop indefinitely
- This particularly affected plugin workflows that loop over signals/images in a group and add results to the same or another group
- The fix ensures `get_object_ids()` now returns a copy of the list, making iteration safe even when modifying the group
- This closes Issue #274 - Infinite loop when iterating over group objects while adding new objects

Marker visibility:

- Fixed cross marker (shown when pressing Alt key on plot) colors inherited from PlotPy being too pale on white background, and other marker properties explicitly configured for better contrast
- Fixed default annotation colors inherited from PlotPy being too pale on white background - segment and drag shape annotations now use brighter green (`#00ff55`) for better visibility

Macro panel layout:

- Fixed macro console taking excessive vertical space on first open - the script editor now properly gets 70% of space and console 30% by default, ensuring comfortable editing without manual resizing

Macro naming:

- Fixed automatic macro naming generating duplicate names after reloading workspace from HDF5 files
- When macros were deserialized from saved workspaces or imported from files, the internal counter used for generating default names (e.g., "macro_01", "macro_02") was not updated, causing the next new macro to potentially reuse an existing name
- The counter now synchronizes with existing macro names after loading, ensuring unique sequential naming
- Default macro names simplified from translated "Untitled XX" to language-neutral "macro_XX" format for better consistency across locales

Macro execution:

- Fixed syntax errors when using f-strings with nested quotes in macros (e.g., `f'text {func("arg")}'` now works correctly)
- Fixed corrupted Unicode characters in macro console output on Windows - special characters like ✓, ⚠, and → now display correctly instead of showing garbled text

Internal console - HDF5 workspace operations:

- Fixed application freeze when calling `open_h5_files()` from the internal console - the console runs in a separate thread and creating Qt GUI elements (progress dialogs) from non-main threads causes deadlocks
- Added new headless API methods `load_h5_workspace()` and `save_h5_workspace()` that can be safely called from the internal console or any script context without GUI dependencies
- These methods are also available through the remote control API for consistency
- Note: `load_h5_workspace()` only supports native DataLab HDF5 files; for importing arbitrary HDF5 files, use macros with `RemoteProxy`
- This closes Issue #275 - Console freezes when calling `open_h5_files()` from internal console

Signal cursor dialogs - Decimal input with regional settings:

- Fixed decimal value input failing in signal cursor dialogs when using regional settings with comma as decimal separator (e.g., French, German locales)
- The Y input field in "First abscissa at y=…", "Ordinate at x=…", and "Full width at y=…" dialogs now consistently accepts dot as decimal separator regardless of system locale
- This closes Issue #276 - Decimal input fails with regional settings using comma as decimal separator

Radial profile title:

- Fixed duplicate suffix in result image title when extracting radial profile from an image (e.g., `radial_profile(i019)|center=(192.500, 192.500)|center=(192.500, 192.500)` instead of `radial_profile(i019)|center=(192.500, 192.500)`)
- This fix is provided by Sigima 1.0.4

Grid ROI - Missing spacing parameters for non-uniform grids:

- Fixed grid ROI feature not working correctly for images where subimages don't fill the entire image area (e.g., laser spot arrays with gaps between spots)
- Added missing `xstep` and `ystep` parameters to control horizontal and vertical spacing between ROI centers, as a percentage of the automatically computed cell width/height (default 100% = evenly distributed grid)
- Previously, the grid was always assumed to be evenly distributed across the entire image, which failed when there was significant offset or gaps between features
- Users can now adjust ROI spacing independently from ROI size to accurately extract grids from real-world data like laser spot arrays, diffraction patterns, or any regularly spaced features with gaps
- This fix is provided by Sigima 1.0.4
- This closes Issue #282 - Grid ROI Missing Spacing Parameters

Analysis auto-recompute - Redundant $O(n^2)$ calculations when adding ROI:

- Fixed severe performance issue where adding ROI to N selected images with analysis results (e.g., Statistics, Centroid) triggered N^2 calculations instead of N
- When 10 images were selected with pre-computed statistics, adding a ROI would show 10 progress bars of 10 calculations each (100 total) instead of a single progress bar with 10 calculations
- The issue occurred because `auto_recompute_analysis()` was calling `compute_1_to_0()` which by default processes all selected objects, not just the specific object being recomputed
- Added `target_objs` parameter to `compute_1_to_0()` to allow specifying which objects to process, and updated `auto_recompute_analysis()` to use it
- This closes Issue #283 - Redundant $O(n^2)$ calculations when adding ROI to multiple images with analysis results

Result visualization - Analysis result segments hard to see:

- Fixed analysis result markers (FWHM, pulse features, etc.) being difficult or impossible to see on signal plots
- These measurement indicators were appearing black on white backgrounds or white on dark backgrounds, making them blend into the plot
- They now display in bright green with thicker lines, making them clearly visible in all color themes
- Also removed the filled area that was obscuring the underlying signal data - markers now show only the boundary lines indicating the measured region

5.5.3 DataLab Version 1.0.2 (2025-12-03)

🛠 Bug Fixes since version 1.0.1

Signal axis calibration - Replace X by other signal's Y:

- Added new "Replace X by other signal's Y" operation in Processing > Axis transformation menu for signal calibration workflows
- Addresses critical missing functionality reported by users who needed to apply wavelength calibration or similar transformations to spectroscopy data
- The operation combines two signals: uses Y values from one signal as the new X coordinates for another signal's Y values
- Unlike X-Y mode (which resamples and interpolates), this operation directly uses Y arrays without interpolation, preserving exact calibration values
- Requires both signals to have the same number of points - raises clear error message if sizes don't match
- Automatically transfers metadata: X axis label/unit taken from calibration signal's Y label/unit
- Menu location: Processing > Axis transformation > "Replace X by other signal's Y"
- This closes Issue #273 - Missing signal axis calibration: no way to replace X with Y from another signal

X-Y mode:

- The X-Y mode processing operation for signals has been moved to Processing > Axis transformation > "X-Y mode" for better discoverability
- The nuance between X-Y mode (which resamples/interpolates) and the new "Replace X by other signal's Y" operation has been clarified in documentation

Lock LUT setting persistence:

- Fixed "Lock LUT range when updating data" setting not persisting in Settings > Visualization > Images > Default Image visualization settings
- The `keep_lut_range` parameter was not being saved to configuration, causing the checkbox to systematically uncheck itself after validation (added missing `ima_def_keep_lut_range` option in configuration)
- This closes Issue #270 - Lock LUT setting not persisting in image visualization defaults

Custom signal creation:

- Fixed `AttributeError: 'NoneType' object has no attribute 'T'` error when creating a custom signal from the menu
- This closes Issue #269 - Custom Signal Creation: `AttributeError` when creating signal from menu

Macro execution:

- Fixed `UnicodeEncodeError` when executing macros that print Unicode characters (e.g., arrows →) on Windows systems with certain locales, e.g. cp1252 (closes Issue #263)
- The macro subprocess now automatically uses UTF-8 encoding for stdout and stderr, eliminating the need to manually add `sys.stdout.reconfigure(encoding='utf-8')` at the beginning of each macro.

ROI coordinate precision:

- ROI coordinates are now automatically rounded to appropriate precision when defining ROIs interactively from geometrical shapes, avoiding excessive decimal places while maintaining reasonable precision relative to data sampling (1/10th of sampling period for signals, 1/10th of pixel spacing for images)
- ROI coordinates are also rounded when displaying them in the "Edit numerically" dialog, preventing floating-point arithmetic errors from showing excessive decimal places (e.g., 172.29999999999995 is now displayed as 172.3)

- This closes Issue #266 - Excessive decimal places in ROI coordinates

Polygonal ROI handling:

- Fixed `ValueError: Buffer has wrong number of dimensions` error when creating masks from polygonal ROIs in the ROI editor
- The PolygonalTool was incorrectly initializing ROI coordinates as a nested list instead of a flat list, causing mask computation to fail

HDF5 file opening dialog:

- Fixed bug where user's choice in the "clear workspace" confirmation dialog was ignored when opening HDF5 files
- When the user clicked "No" in the dialog, the default configuration setting was applied instead of respecting the user's choice
- This closes Issue #267 - HDF5 file opening dialog ignores user's choice

Creation tab axis update:

- Fixed plot not updating when modifying only xmin/xmax parameters for distribution signals (Zero, Normal, Poisson, Uniform) in the Creation tab
- The issue occurred because the data hash calculation only considered Y values, so changes to X axis bounds were not detected
- Plot now properly refreshes when any axis parameter changes, even if Y values remain identical
- This closes Issue #268 - Creation tab axis not updating for distribution signals

ROI statistics with out-of-bounds ROI:

- Fixed `ValueError: zero-size array to reduction operation minimum which has no identity` error when computing statistics on images with ROI extending beyond canvas boundaries
- The issue occurred when a ROI partially or completely extended outside the image bounds, resulting in empty array slices during statistics computation
- ROI bounding boxes are now properly clipped to image boundaries, and fully out-of-bounds ROIs return NaN statistics values
- This fix is implemented in Sigima library (see Issue #1 - `ValueError` when computing statistics on ROI extending beyond image boundaries)

Object property panel tab selection:

- Fixed tab selection behavior in object properties panel to be more predictable and user-friendly
- Properties tab is now always shown by default when switching between objects, providing consistent navigation
- Creation, Processing, and Analysis tabs now appear automatically only once after their respective triggering events (object creation, 1-to-1 processing, or analysis computation), then revert to Properties tab for subsequent selections
- This eliminates the confusing behavior where the tab would arbitrarily persist or change based on previous selections
- This closes Issue #271 - Improve object property panel tab selection behavior

5.5.4 DataLab Version 1.0.1 (2025-11-18)

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 (released 2025-11-16), 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.

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

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

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

◿ 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)

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

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

🔧 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)

🛠 Bug Fixes since version 0.20

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

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

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)

5.6 Version 0.20

5.6.1 DataLab Version 0.20.1 (2025-07-15)

🛠 Bug fixes:

- Fixed Issue #233 - Hard crash when trying to activate the curve stats tool on a zero signal
- Fixed Issue #184 - Curve marker style unexpectedly changes to "Square" after validating "Parameters…" dialog
- Fixed Issue #117 - DataLab's signal moving median crashes on Linux with `mode='mirror': free(): invalid next size (normal)` (this is a bug in SciPy v1.15.0 to v1.15.2, which was fixed in SciPy v1.15.3)
- Fixed Issue #186 - Image text files with a comma as decimal separator cannot be opened directly (only the Import Wizard does support this)
- Fixed Issue #238 - Image text files with a trailing delimiter leads to data with a superfluous column when opened directly (does not happen in the Import Wizard)
- Fixed Issue #239 - Text Import Wizard does not preserve user-defined titles and units
- Fixed Issue #240 - Text Import Wizard does not preserve user-defined data type (e.g. `int16`, `float32`, etc.)
- Fixed Issue #235 - Text Import Wizard: add support for importing signal files with integer values
- Fixed Issue #236 - Text Import Wizard: add support for `.mca` files
- Fixed Issue #243 - View in a new window (image): intensity profile tools are sometimes disabled (fixed in PlotPy v2.7.5)

5.6.2 DataLab Version 0.20.0 (2025-04-28)

✨ 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`

5.7 Version 0.19

5.7.1 DataLab Version 0.19.2 (2025-04-28)

🛠 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)

5.7.2 DataLab Version 0.19.1 (2025-04-08)

🛠 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

5.7.3 DataLab Version 0.19.0 (2025-03-31)

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

5.8 Version 0.18

5.8.1 DataLab Version 0.18.2 (2025-03-16)

ⓘ 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

5.8.2 DataLab Version 0.18.1 (2024-12-13)

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

5.8.3 DataLab Version 0.18.0 (2024-11-14)

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.

5.9 Version 0.17

5.9.1 DataLab Version 0.17.1 (2024-10-01)

ⓘ 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

5.9.2 DataLab Version 0.17.0 (2024-08-02)

ⓘ 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

5.10 Version 0.16

5.10.1 DataLab Version 0.16.4 (2024-07-09)

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

5.10.2 DataLab Version 0.16.3 (2024-07-03)

🛠 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

5.10.3 DataLab Version 0.16.2 (2024-07-01)

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)

5.10.4 DataLab Version 0.16.1 (2024-06-21)

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

5.10.5 DataLab Version 0.16.0 (2024-06-13)

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)

5.11 Version 0.15

5.11.1 DataLab Version 0.15.1 (2024-05-03)

🛠 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

5.11.2 DataLab Version 0.15.0 (2024-04-11)

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)

5.12 Version 0.14

5.12.1 DataLab Version 0.14.2 (2024-03-22)

⚠ 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

5.12.2 DataLab Version 0.14.1 (2024-03-18)

🎉 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

5.12.3 DataLab Version 0.14.0 (2024-03-05)

🔅 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

5.13 Version 0.12

5.13.1 DataLab Version 0.12.0 (2024-02-16)

 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>/σ(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

5.14 Version 0.11

5.14.1 DataLab Version 0.11.0 (2024-01-23)

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

5.15 Version 0.10

5.15.1 DataLab Version 0.10.1 (2023-12-22)

Note: V0.10.0 (released 2023-12-22) 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`

5.16 Version 0.9

5.16.1 DataLab Version 0.9.2 (2023-12-15)

🛠 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

5.16.2 DataLab Version 0.9.1 (2023-12-01)

🛠 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

5.16.3 DataLab Version 0.9.0 (2023-11-29)

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.