DataLab

Release 0.12.0

Feb 16, 2024

CONTENTS

DataLab is an **open-source platform for scientific and technical data processing and visualization** with unique features designed to meet industrial requirements. Leveraging the richness of the scientific Python ecosystem[1] and the Qt graphical user interfaces, DataLab is a versatile tool, extendable with *Plugins* and working seamlessly with *your IDE* or *your Jupyter notebooks*.

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

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

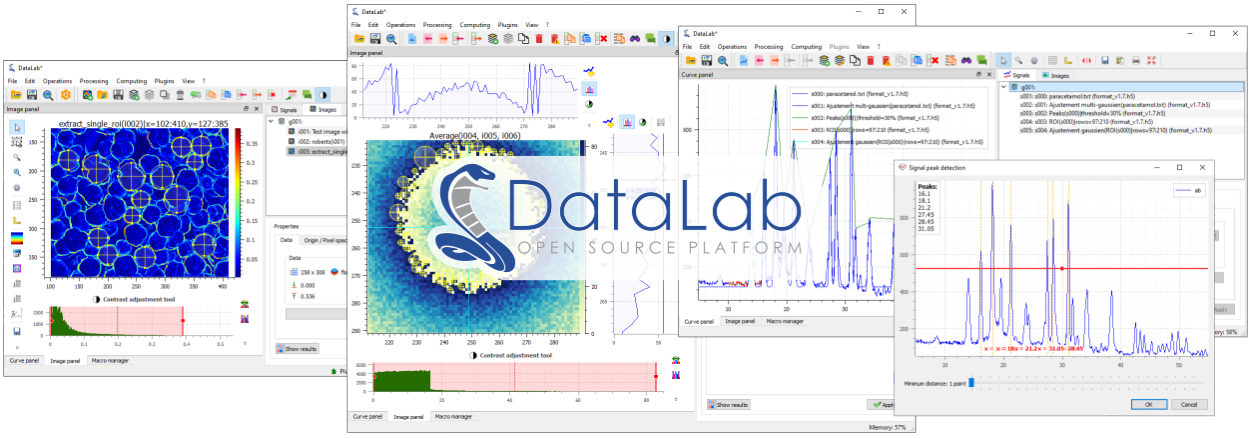


Fig. 1: Signal and image visualization in DataLab

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.

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

[1] DataLab processing features are mainly based on NumPy, SciPy, scikit-image, OpenCV and PyWavelets libraries. DataLab visualization capabilities are based on PlotPyStack toolkit, a set of Python libraries for building scientific applications with Qt graphical user interfaces.

CHAPTER

ONE

GETTING STARTED

DataLab is an open platform for signal and image processing. Its functional scope is intentionally broad. With its many functions, some of them technically advanced, DataLab enables the processing and visualization of all types of scientific data. As a result, scientific, industrial, and innovation stakeholders can have access to an easy-to-use tool that is simple to adapt and offers the reliability of industrial-grade software.

1.1 Installation

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

See also:

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

1.1.1 How to install

DataLab is available in several forms:

- As a 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!
- As a precompiled *Wheel package*, which can be installed using `pip`.
- As a *Source package*, which can be installed using `pip` or manually.

See also:

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

Package manager `pip`

GNU/Linux Windows macOS

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

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 cdl[qt]
```

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

```
$ pip install cdl
```

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 cdl[qt]
```

All-in-one installer

Windows

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

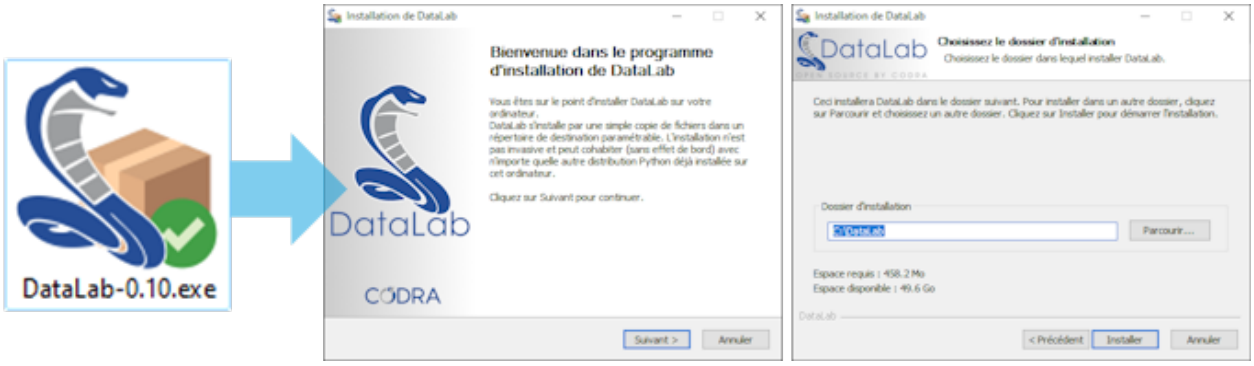


Fig. 1: DataLab all-in-one installer for Windows

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

Warning: DataLab Windows installer is available for Windows 8, 10 and 11 (main release, based on Python 3.11) and also for Windows 7 SP1 (Python 3.8 based release, see file ending with `-Win7.exe`).

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.

Wheel package

GNU/Linux Windows macOS

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

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

Source package

GNU/Linux Windows macOS

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

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

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

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

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

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

Development version

GNU/Linux Windows macOS

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

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

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

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

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

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

1.1.2 Dependencies

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

The `cdl` package requires the following Python modules:

Name	Version	Summary
Python	>=3.8, <4	
h5py	>= 3.0	
NumPy	>= 1.21	
SciPy	>= 1.7	
scikit-image	>= 0.18	
opencv-python-headless	>= 4.5	
PyWavelets	>= 1.1	
psutil	>= 5.5	
guidata	>= 3.2	
PlotPy	>= 2.0	
QtPy	>= 1.9	
PyQt5	>=5.11	Python bindings for the Qt cross platform application toolkit

Optional modules for development:

Name	Version	Summary
black		The uncompromising code formatter.
isort		A Python utility / library to sort Python imports.
pylint		python code static checker
Coverage		Code coverage measurement for Python
pyinstaller	>=6.0	PyInstaller bundles a Python application and all its dependencies into a single package.

Optional modules for building the documentation:

Name	Version	Summary
PyQt5		Python bindings for the Qt cross platform application toolkit
sphinx		Python documentation generator
sphinx_intl		Sphinx utility that make it easy to translate and to apply translation.
sphinx-sitemap		Sitemap generator for Sphinx
myst_parser		An extended [CommonMark](https://spec.commonmark.org/) compliant parser,
sphinx_design		A sphinx extension for designing beautiful, view size responsive web components.
sphinx-copybutton		Add a copy button to each of your code cells.
pydata-sphinx-theme		Bootstrap-based Sphinx theme from the PyData community

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

1.2 Use cases, main features and key strengths

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

1.2.1 What are the applications for Datalab?

Real world examples

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

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

Usage modes

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

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

Use cases

See also:

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

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

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

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

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

Companion tool for scientific/technical work

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

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

Prototyping a data processing application

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

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

Debugging a data processing application

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

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

Note: DataLab can also be controlled from your familiar development environment (e.g., Visual Studio Code, Spyder, …) or from a Jupyter notebook, in order to perform calculations using your processing functions while leveraging the advanced features of DataLab. See the tutorials *Prototyping a custom processing pipeline* or *DataLab and Spyder: a perfect match* for examples of use.

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

1.2.2 Main features

The main technical features of DataLab include:

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

1.2.3 Key strengths

To summarize, the four key strengths of DataLab are:

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

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

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

Maintainability and testability
: DataLab is an industrial-grade scientific and technical processing software. The built-in automated tests in DataLab cover 90% of its features, which is significant for software with graphical interfaces and helps mitigate regression risks.

1.3 Key features

This page presents briefly DataLab key features.

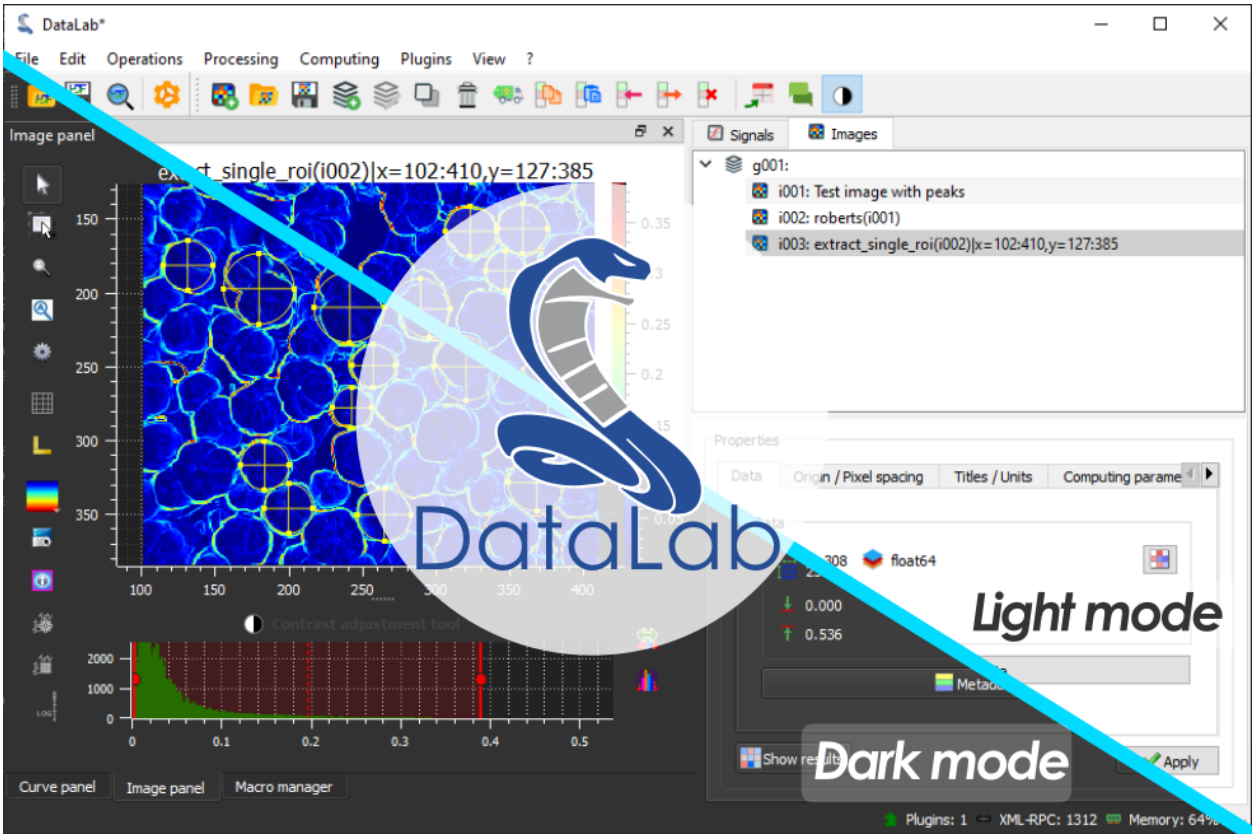


Fig. 2: DataLab supports dark and light mode depending on your platform settings (this is handled by the guidata package, and may be overridden by setting the *QT_COLOR_MODE* environment variable to *dark* or *light*).

1.3.1 Data visualization

Signal	Image	Feature
✓	✓	Screenshots (save, copy)
✓	Z-axis	Lin/log scales
✓	✓	Data table editing
✓	✓	Statistics on user-defined ROI
✓	✓	Markers
	✓	Aspect ratio (1:1, custom)
	✓	50+ available colormaps
	✓	X/Y raw/averaged profiles
✓	✓	Annotations
✓	✓	Persistance of settings in workspace
	✓	Distribute images on a grid

1.3.2 Data processing

Signal	Image	Feature
✓	✓	Process isolation for running computations
✓	✓	Remote control from Jupyter, Spyder or any IDE
✓	✓	Remote control from a third-party application
✓	✓	Sum, average, difference, product, …
✓	✓	ROI extraction, Swap X/Y axes
✓		Semi-automatic multi-peak detection
✓		Convolution
	✓	Flat-field correction
	✓	Rotation (flip, rotate), resize, …
	✓	Intensity profiles (line, average, radial)
	✓	Pixel binning
✓		Normalize, derivative, integral
✓	✓	Linear calibration
	✓	Thresholding, clipping
✓	✓	Gaussian filter, Wiener filter
✓	✓	Moving average, moving median
✓	✓	FFT, inverse FFT
✓		Interpolation, resampling
✓		Detrending
✓		Interactive fit: Gauss, Lorenzt, Voigt, polynomial
✓		Interactive multigaussian fit
	✓	Butterworth filter
	✓	Exposure correction (gamma, log, …)
	✓	Restauration (Total Variation, Bilateral, …)
	✓	Morphology (erosion, dilation, …)
	✓	Edges detection (Roberts, Sobel, …)
✓	✓	Computing on custom ROI
✓		FWHM, FW @ $1/e^2$
	✓	Centroid (robust method w/r noise)
	✓	Minimum enclosing circle center
	✓	2D peak detection

continues on next page

Table 1 – continued from previous page

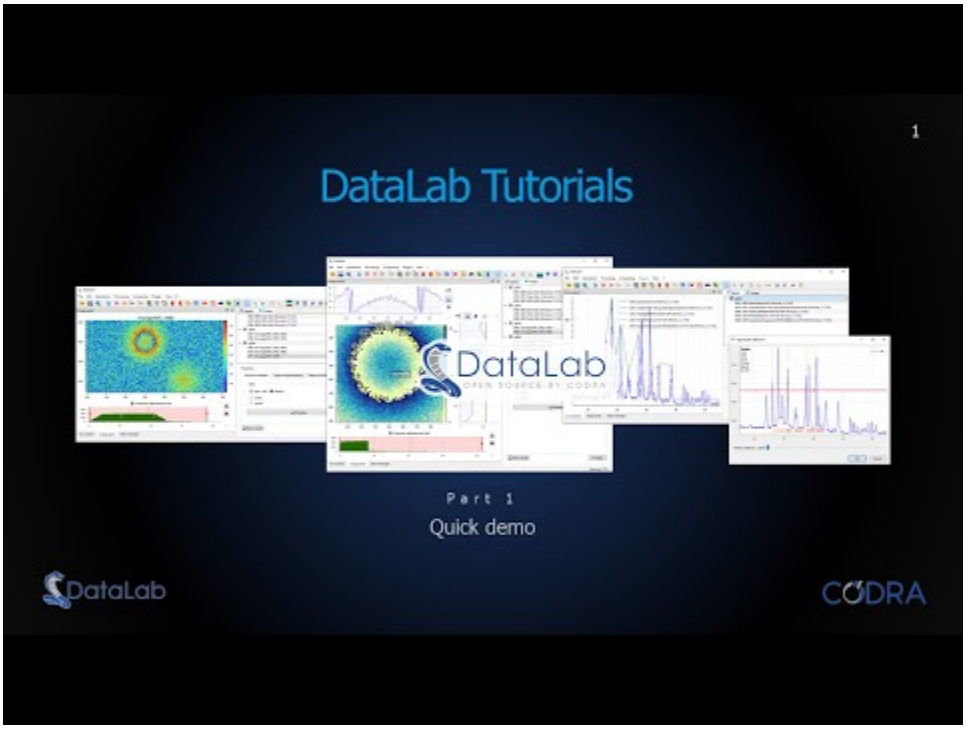
Signal	Image	Feature
	✓	Contour detection
	✓	Circle Hough transform
	✓	Blob detection (OpenCV, Laplacian of Gaussian, ...)

1.4 Tutorials

1.4.1 Video Tutorials

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

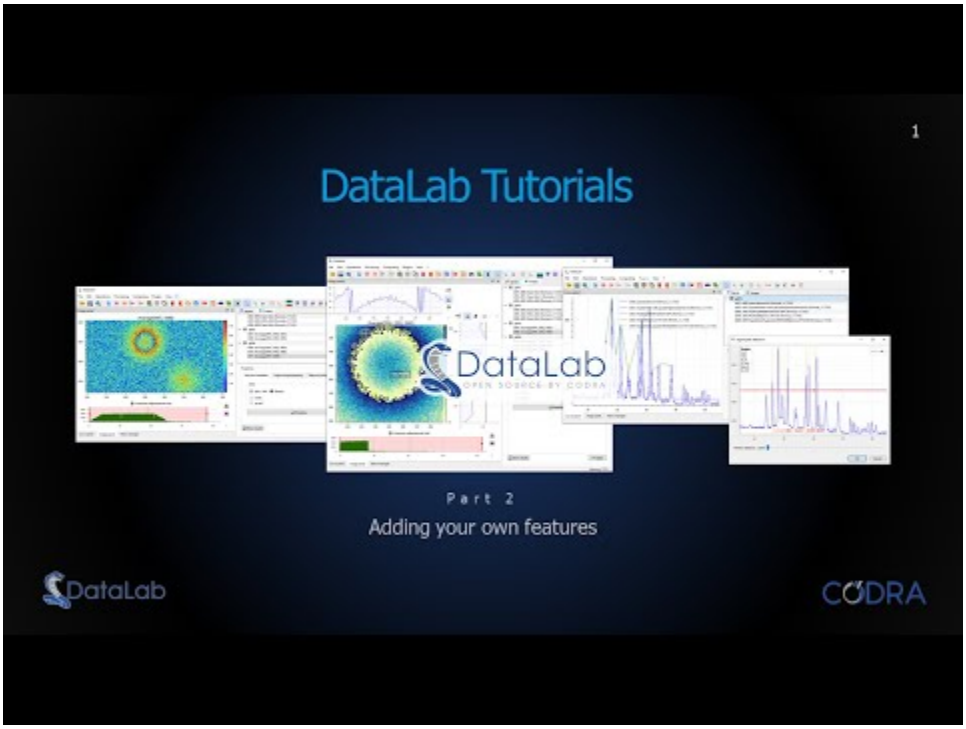
Quick demo



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.

Warning: 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



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

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

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

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

1.4.2 Other Tutorials

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

Processing a spectrum

This example shows how to process a spectrum with DataLab:

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

First, we open DataLab and read the spectrum from a file.

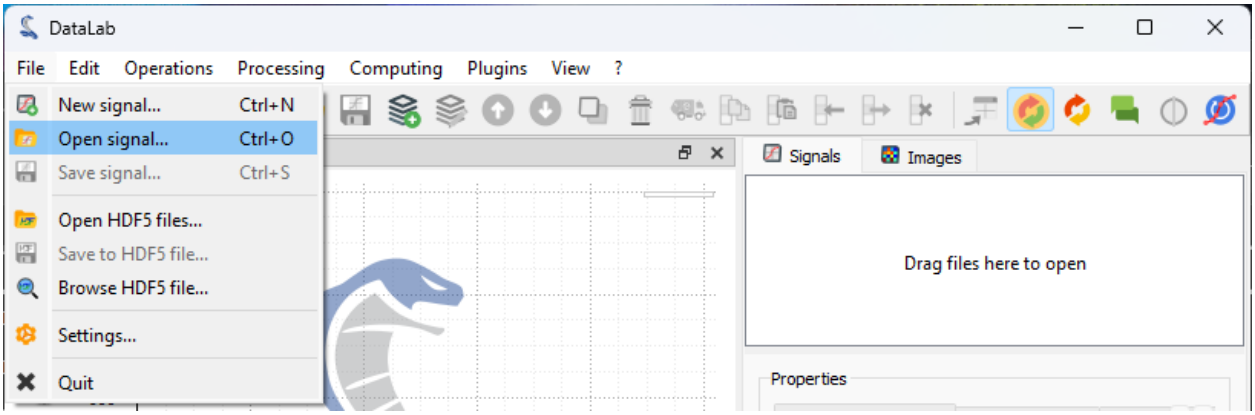


Fig. 3: Open the spectrum file with "File > Open…" , or with the button in the toolbar, or by dragging and dropping the file into DataLab (on the panel on the right).

Here, we are actually generating the signal from a test data file (using "Plugins > Test data > Load spectrum of paracetamol"), but the principle is the same.

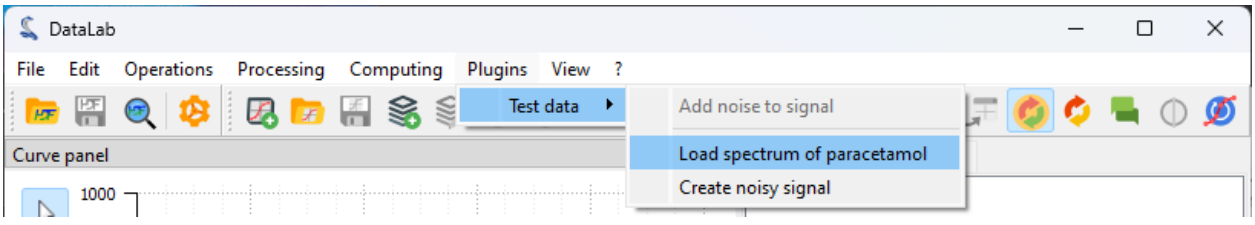


Fig. 4: Using the "Test data" plugin is a convenient way to generate test data for tutorials, but you can use any file containing a spectrum, such as a spectrum from a real experiment.

The spectrum is displayed in the main window.

Now, let's process this spectrum by applying a filter to it. We will use a Wiener filter, which is a filter that can be used to remove noise from a signal, even if this is not absolutely necessary in this case.

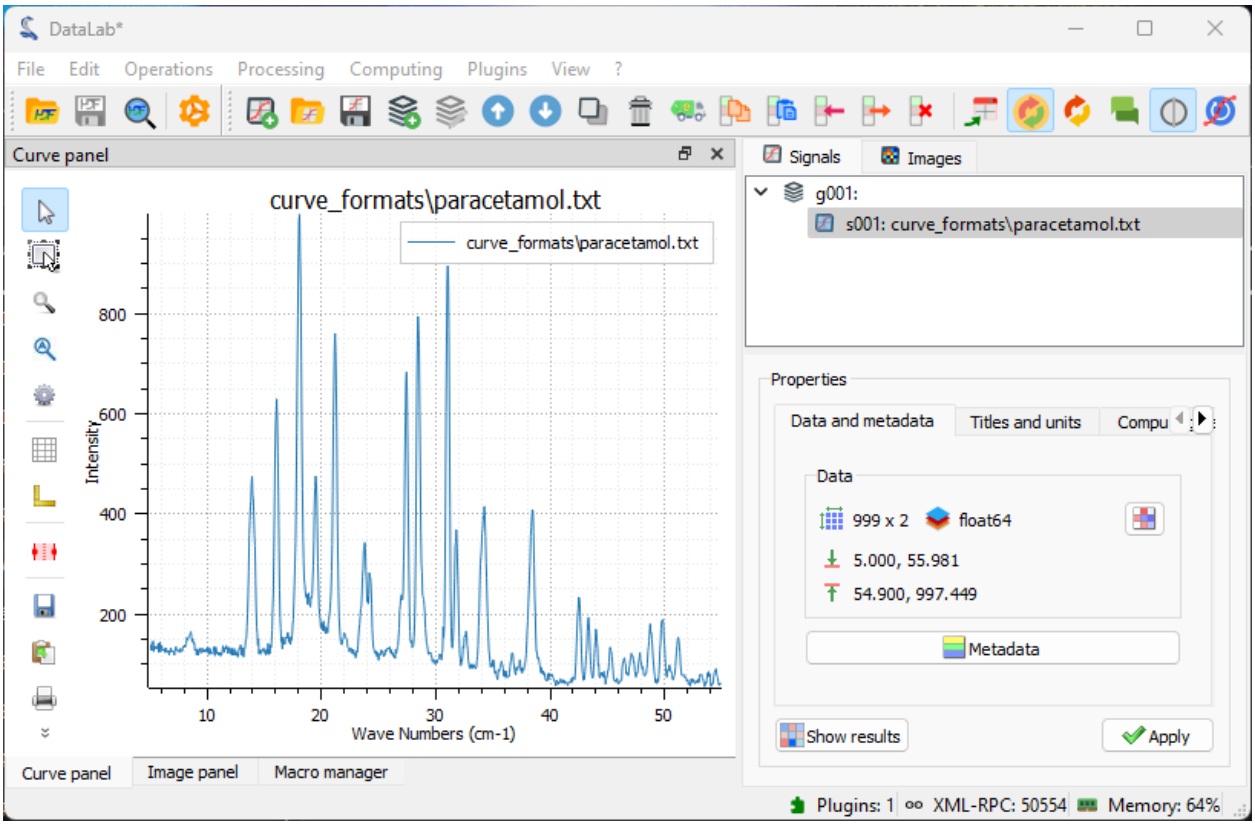


Fig. 5: The spectrum is a 1D signal, so it is displayed as a curve. The horizontal axis is the energy axis, and the vertical axis is the intensity axis.

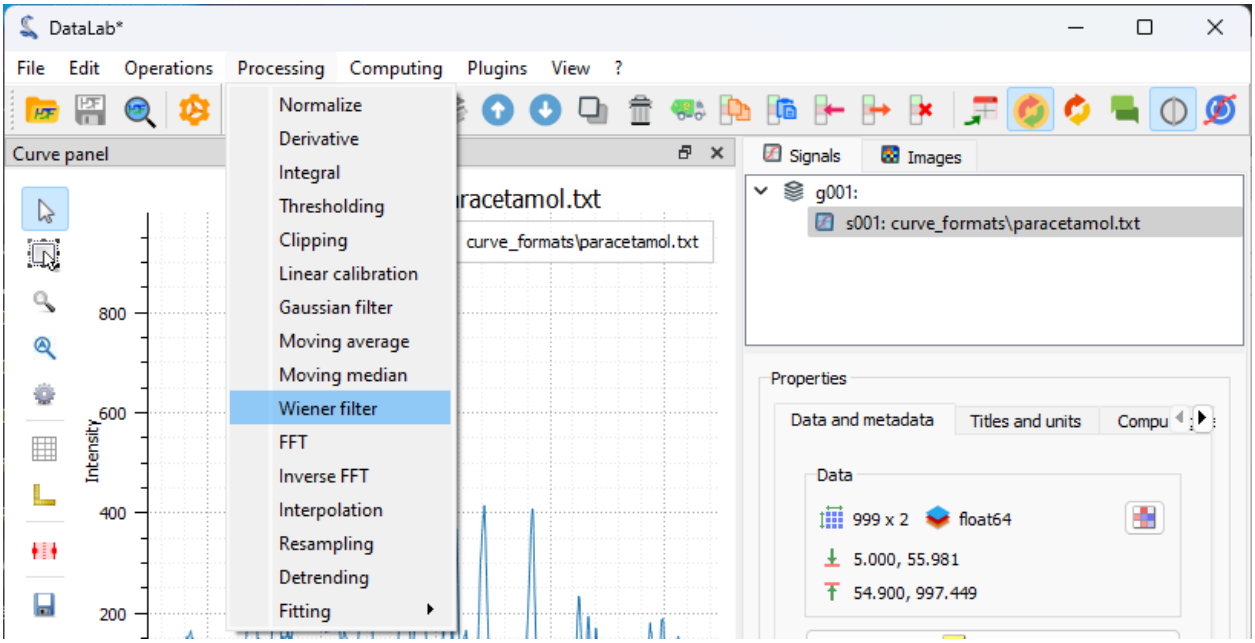


Fig. 6: Open the filter window with “Processing > Wiener filter”.

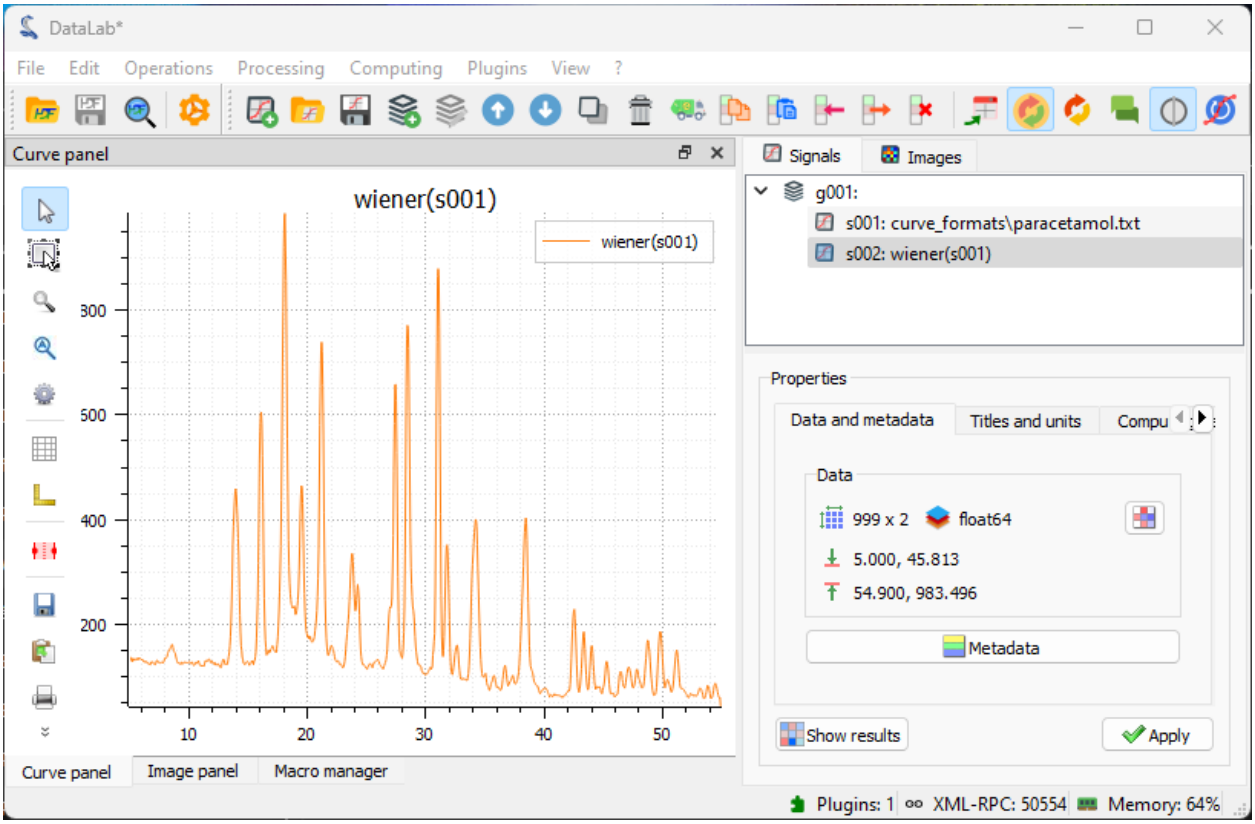


Fig. 7: The result of the filter is displayed in the main window.

If we want to analyze a specific region of the spectrum, we can extract it from the spectrum using the "ROI extraction" feature from the "Operations" menu.

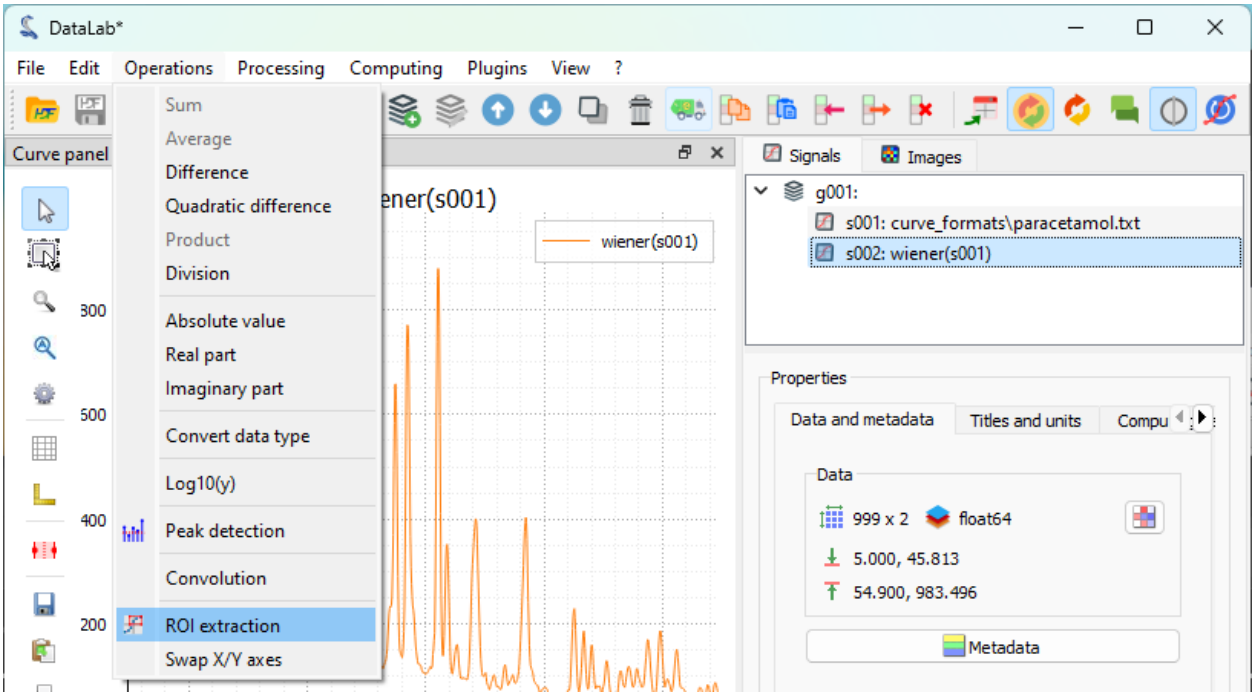


Fig. 8: Open the ROI extraction window with "Operations > ROI extraction".

Let's try to fit a model to the spectrum. We will use a Gaussian model, which is a model that can be used to fit a peak in a spectrum.

To demonstrate another processing feature, we can also try to detrend the spectrum.

When analyzing a spectrum, it can be useful to try to identify the peaks in the spectrum. We can do this by fitting a multi-Gaussian model to the spectrum, using the "Processing > Fitting > Multi-Gaussian fit" feature.

We also could have used the "Peak detection" feature from the "Operations" menu to detect the peaks in the spectrum.

Finally, we can save the workspace to a file. The workspace contains all the signals that were loaded in DataLab, as well as the processing results. It also contains the visualization settings (curve colors, etc.).

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

Detecting blobs on an image

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

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

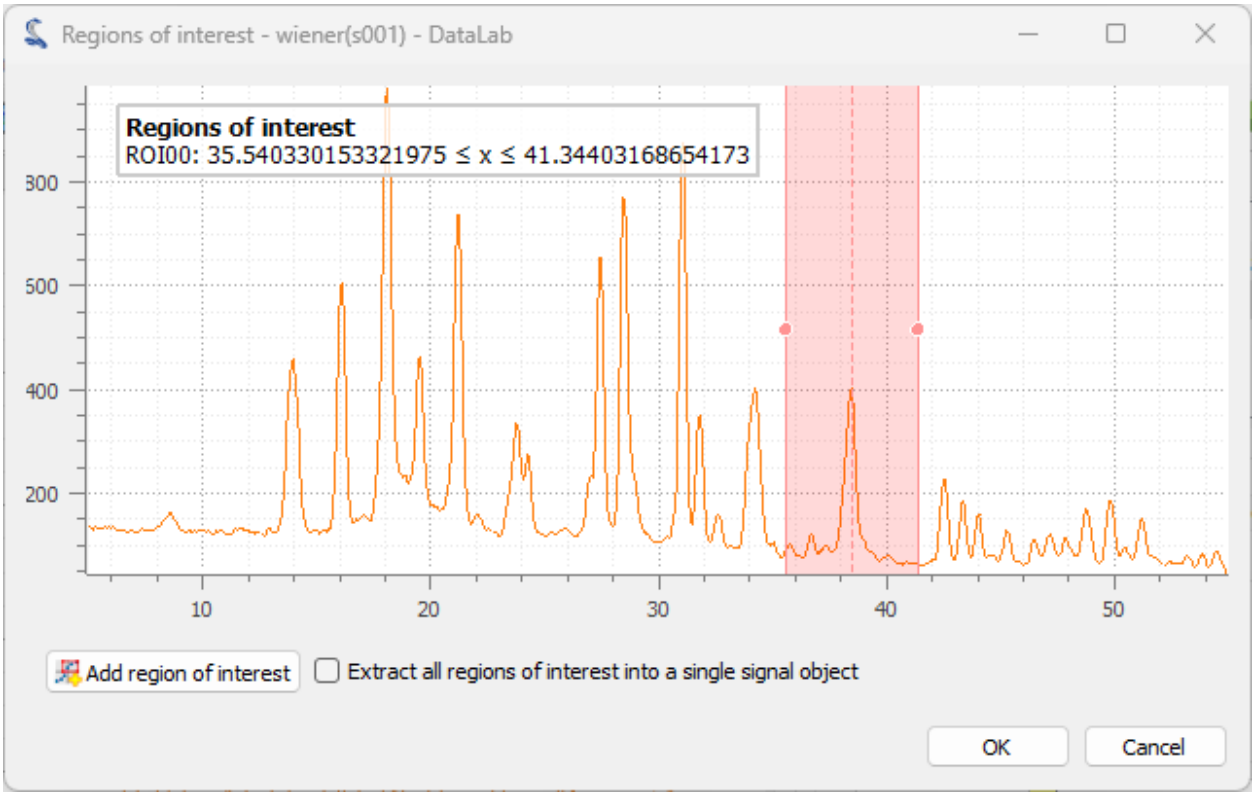


Fig. 9: The “Regions of interest” dialog box is displayed. Click on “Add region of interest” and resize the horizontal window to select the area. Then, click on “OK”.

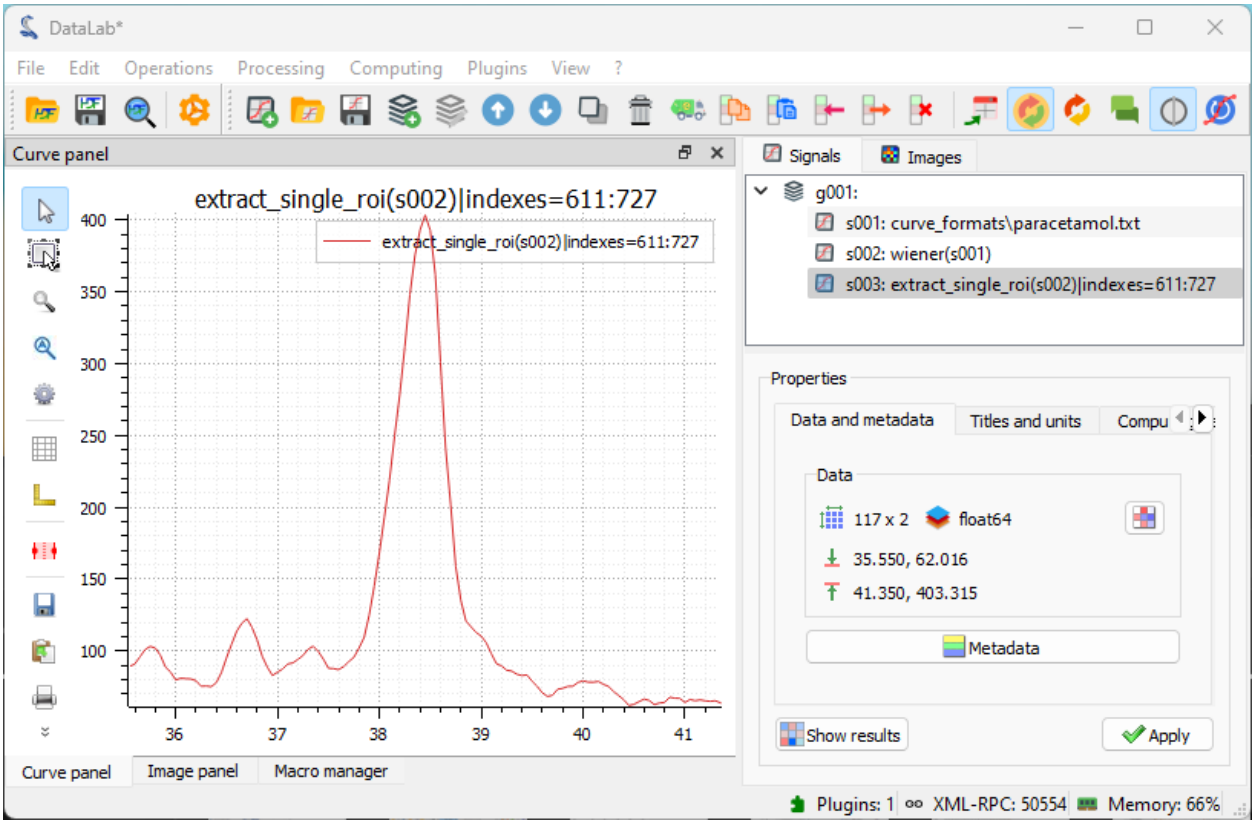


Fig. 10: The region of interest is displayed in the main window.

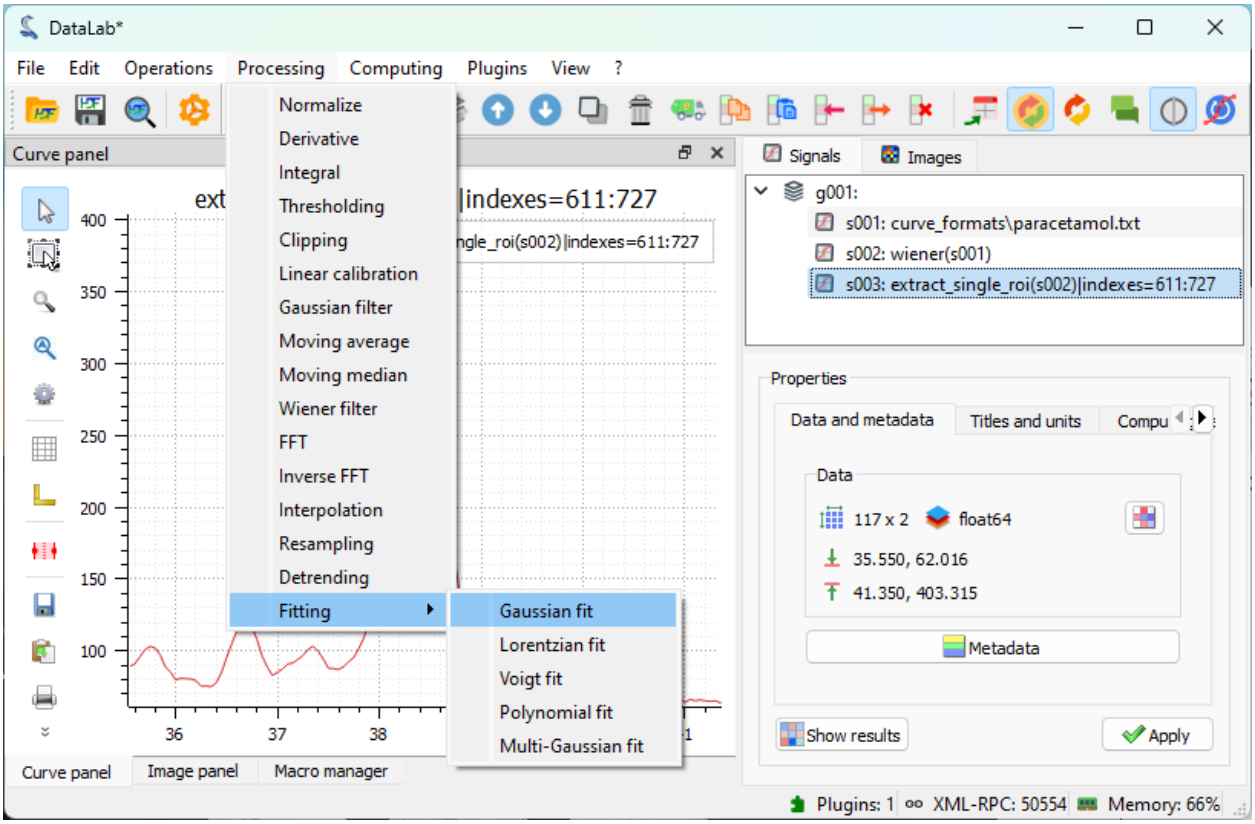


Fig. 11: Open the model fitting window with “Processing > Fitting > Gaussian fit”.

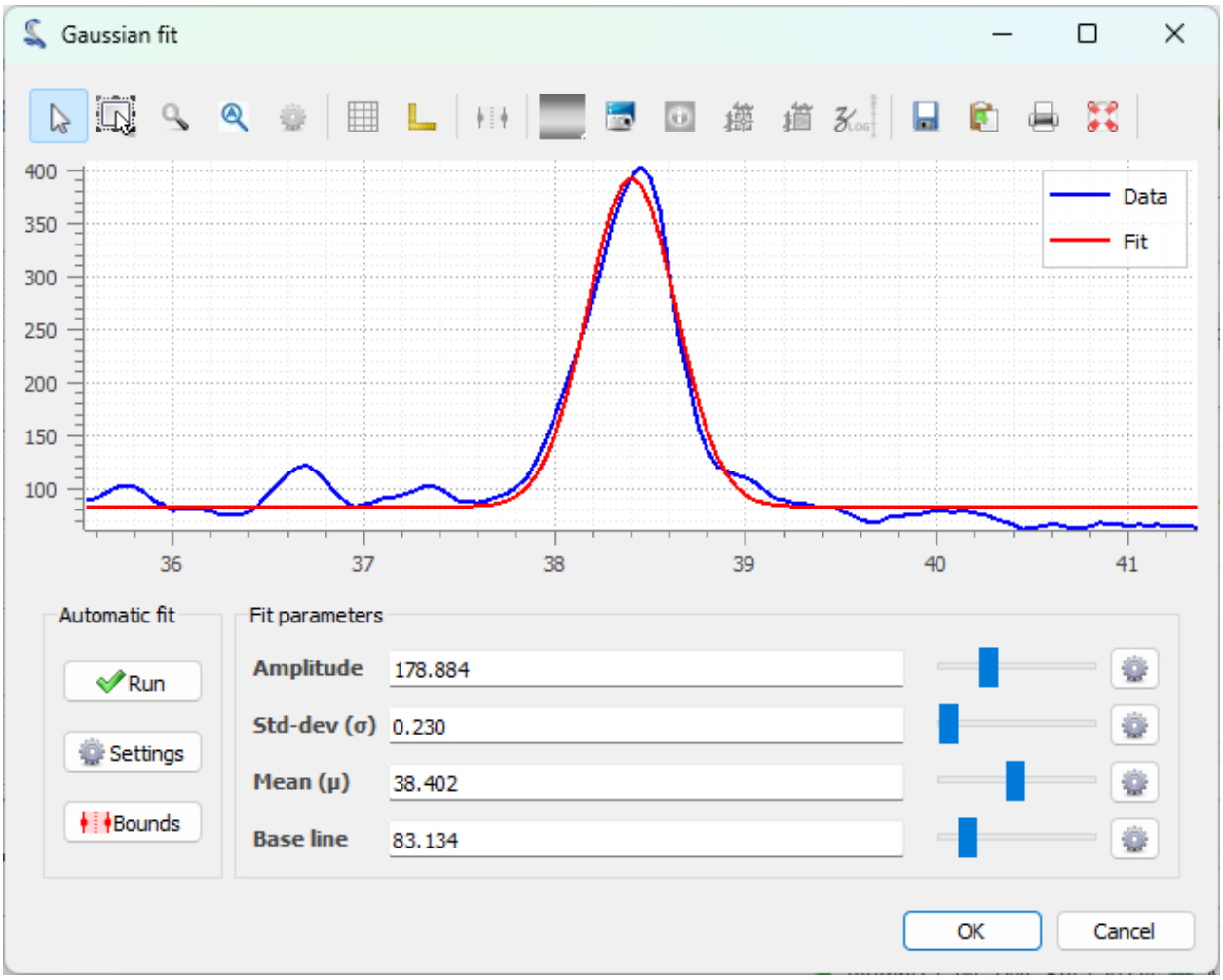


Fig. 12: The "Gaussian fit" dialog box is displayed. An automatic fit is performed by default. Click on "OK" (or eventually try to fit the model manually by adjusting the parameters or the sliders, or try to change the automatic fitting parameters).

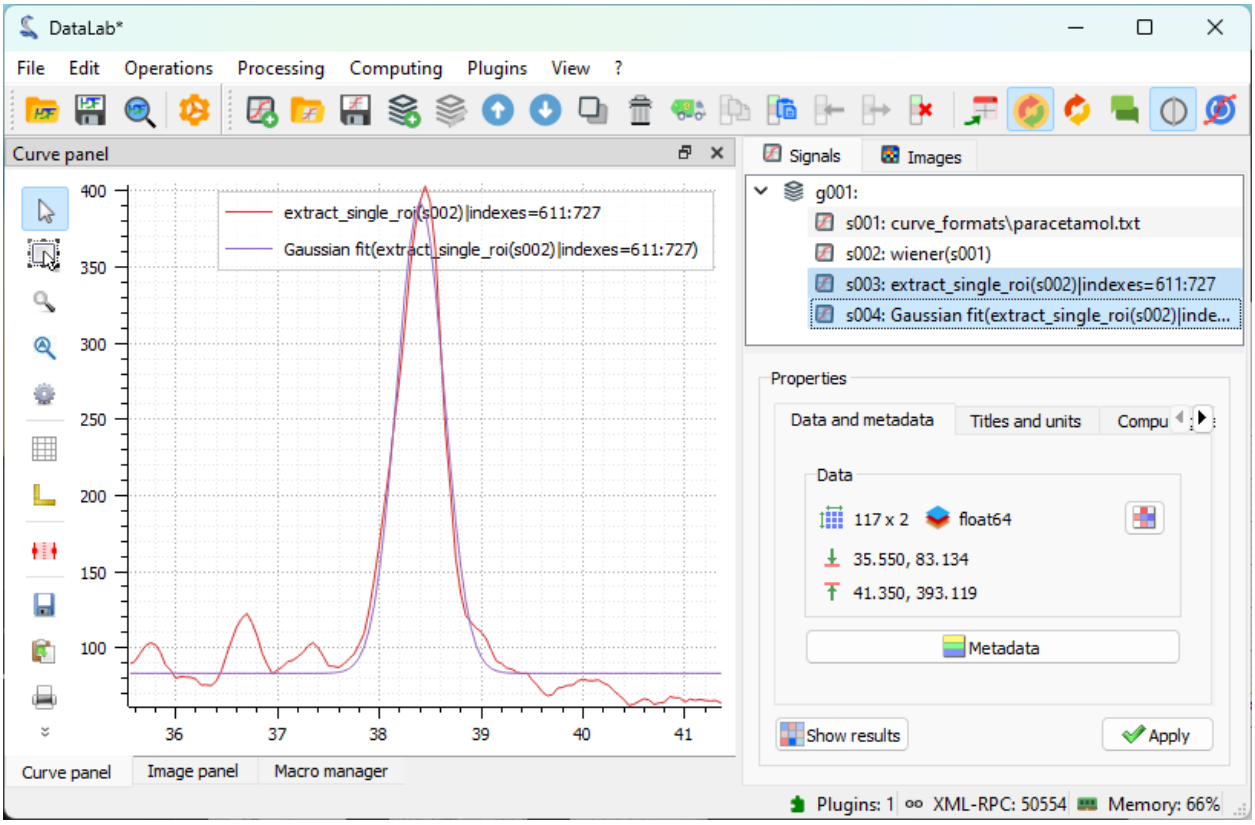


Fig. 13: The result of the fit is displayed in the main window. Here we selected both the spectrum and the fit in the "Signals" panel on the right, so both are displayed in the visualization panel on the left.

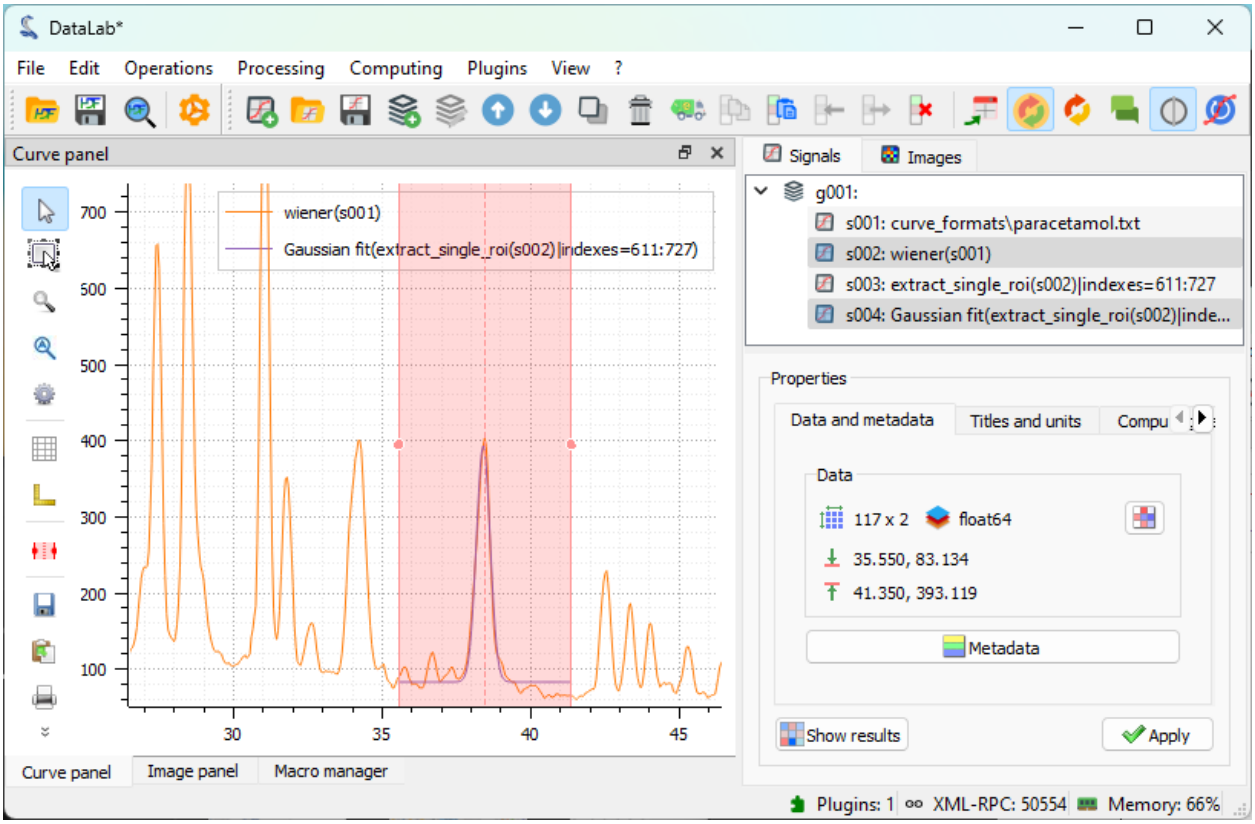


Fig. 14: We may also select the full spectrum and the fit in the "Signals" panel on the right, so that both are displayed in the visualization panel on the left, if this has a sense for the analysis we want to perform. Note that the full spectrum visualization also contains the region of interest we extracted previously.

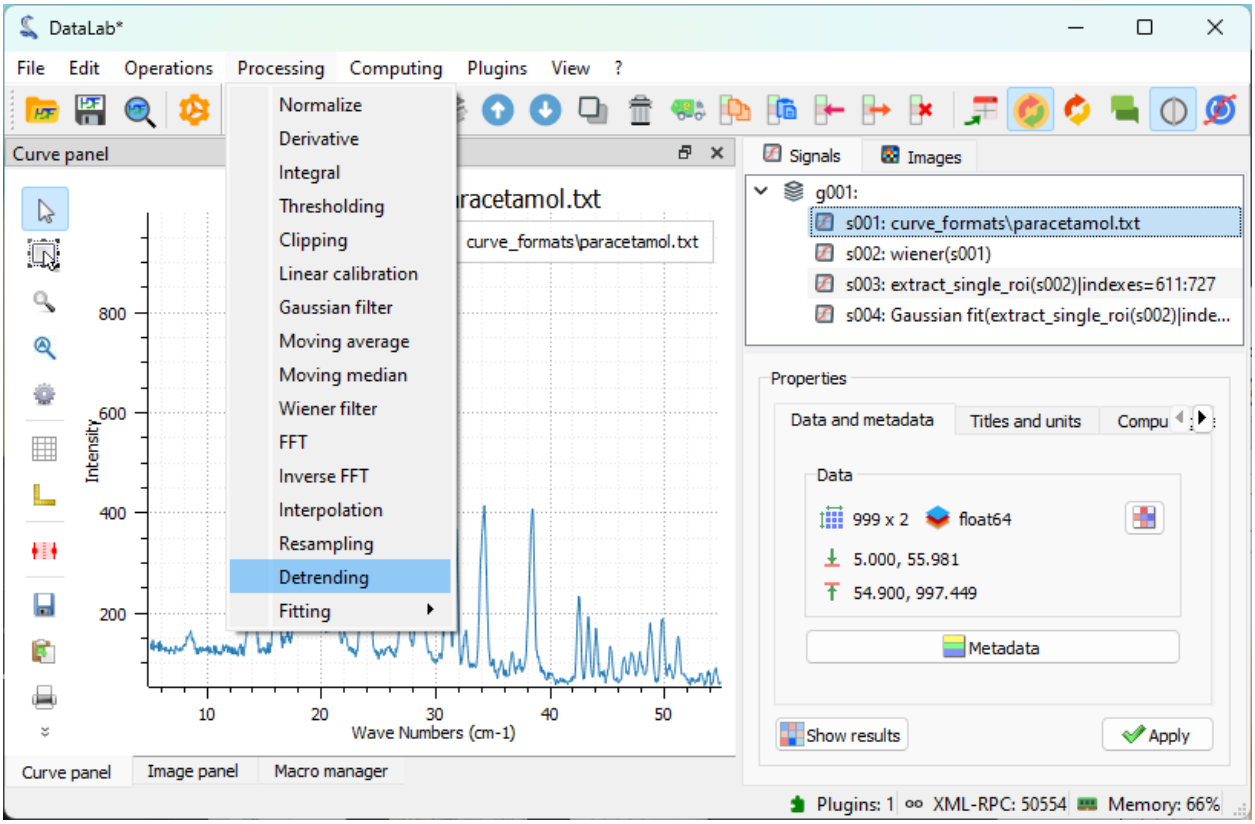


Fig. 15: Execute the “Processing > Detrending” feature.

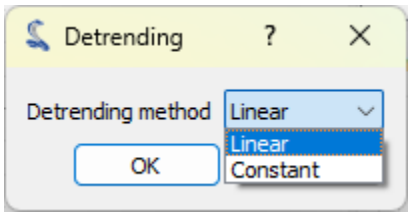


Fig. 16: We choose a linear detrending method, and we click on “OK”.

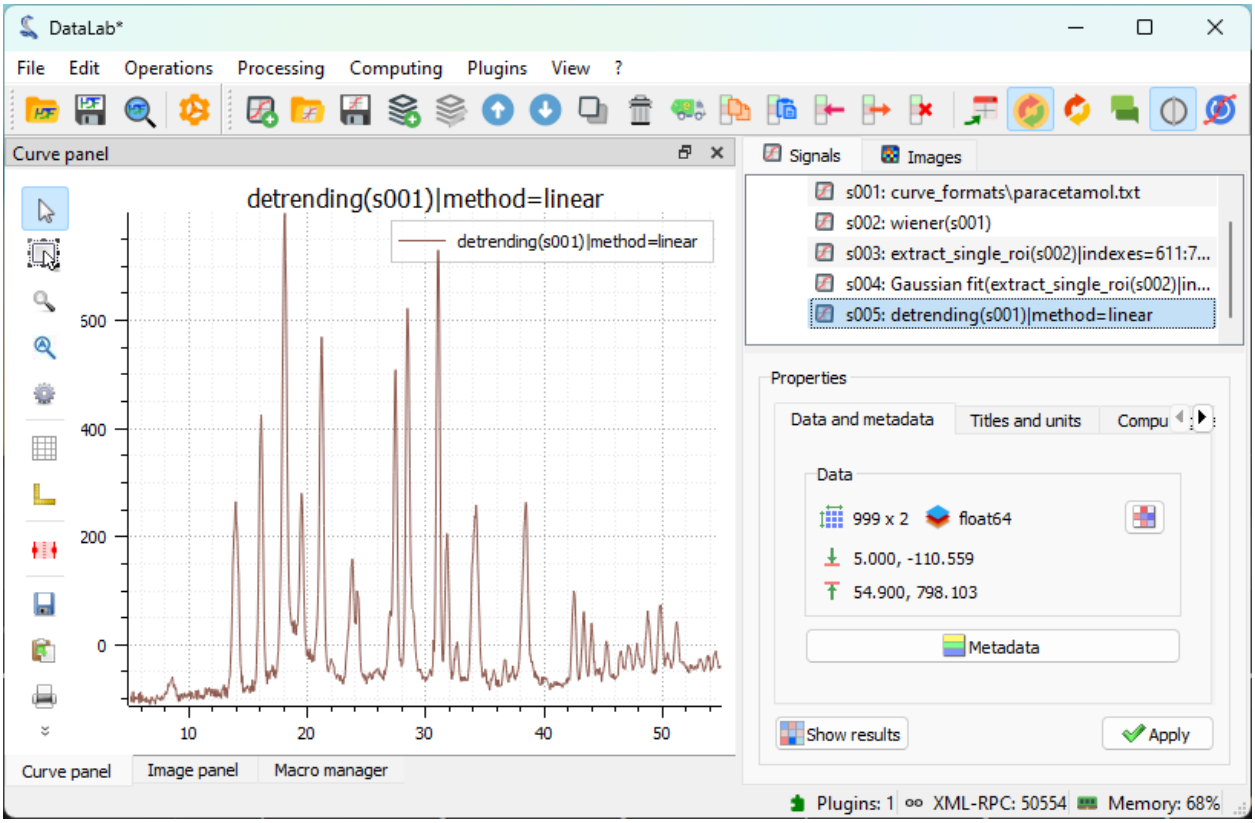


Fig. 17: The result of the detrending is displayed in the main window (in that specific case, the detrending may not be appropriate, but it is just to demonstrate the feature).

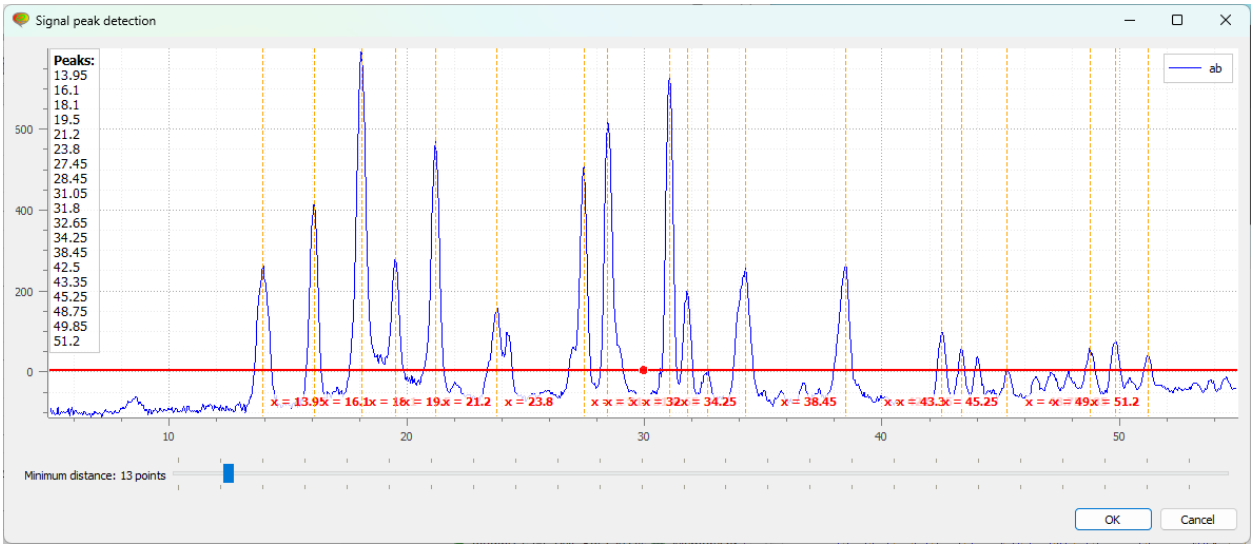


Fig. 18: First, a “Signal peak detection” dialog box is displayed. We can adjust the the vertical cursor position to select the threshold for the peak detection, as well as the minimum distance between two peaks. Then, we click on “OK”.

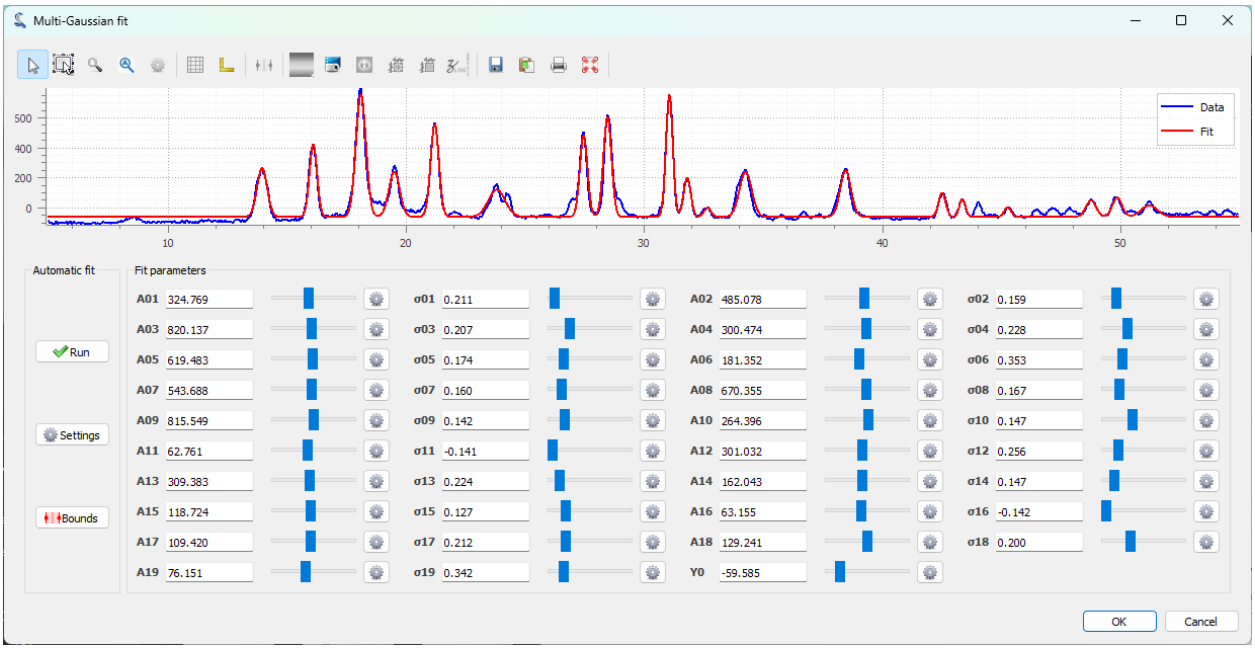


Fig. 19: The "Multi-Gaussian fit" dialog box is displayed. An automatic fit is performed by default. Click on "OK" (or eventually try to fit the model manually by adjusting the parameters or the sliders, or try to change the automatic fitting parameters).

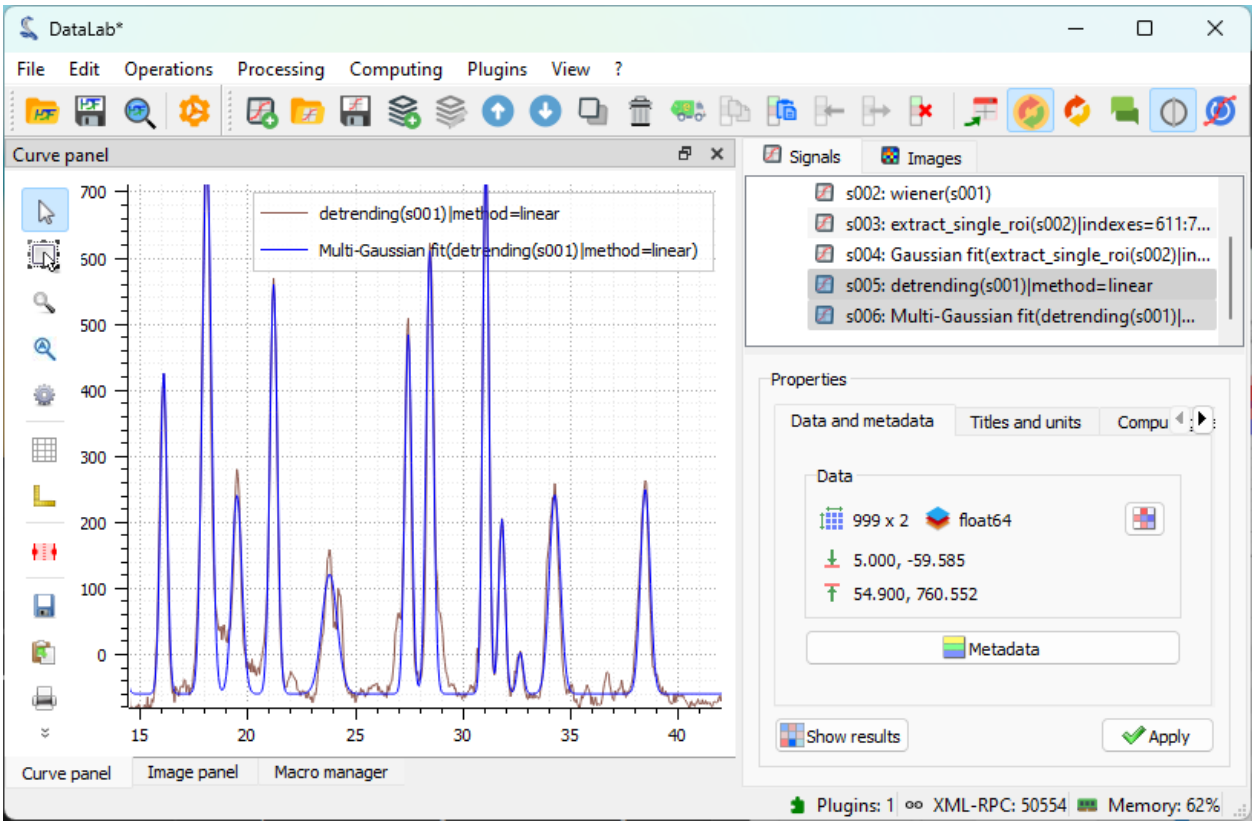


Fig. 20: The result of the fit is displayed in the main window. Here we selected both the spectrum and the fit in the "Signals" panel on the right, so both are displayed in the visualization panel on the left.

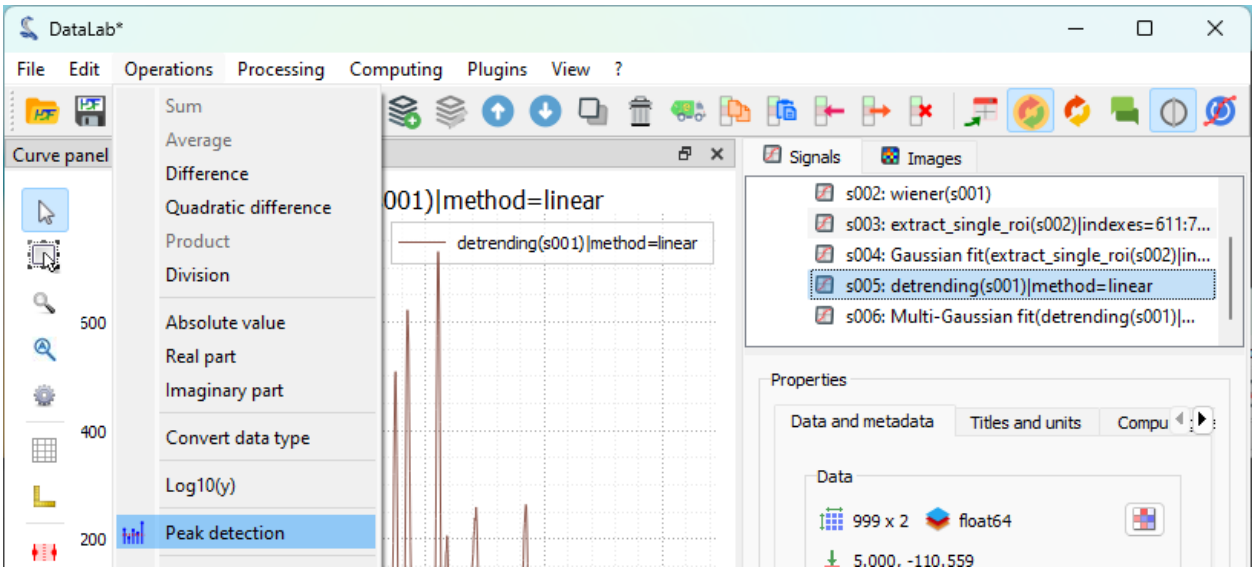


Fig. 21: Open the "Peak detection" window with "Operations > Peak detection".

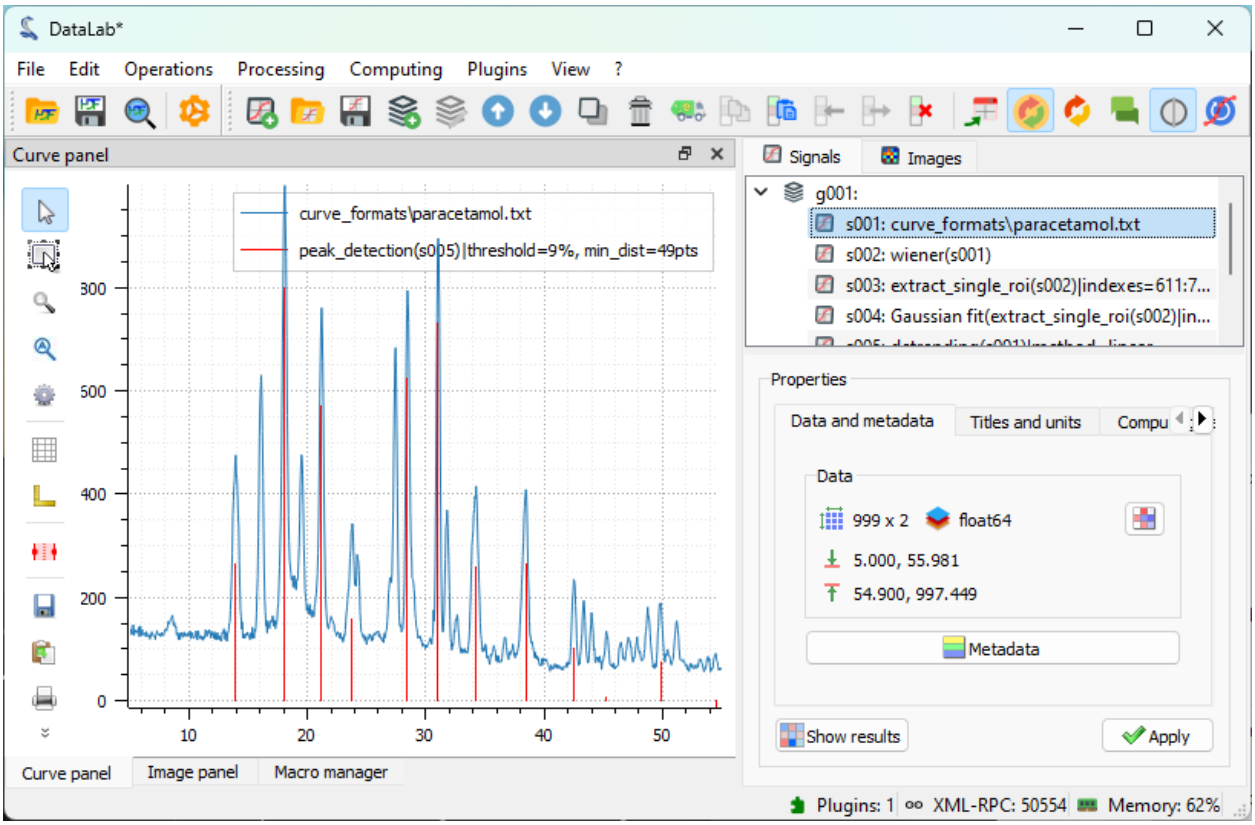


Fig. 22: After having adjusted the parameters of the peak detection dialog (same dialog as the one used for the multi-Gaussian fit), click on "OK". Then, we select the "peak_detection" and the original spectrum in the "Signals" panel on the right, so that both are displayed in the visualization panel on the left.

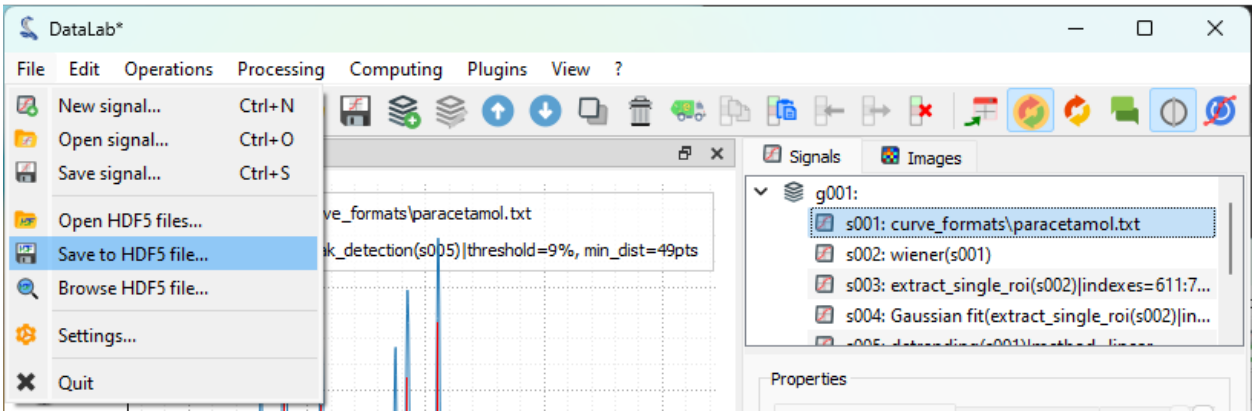


Fig. 23: Save the workspace to a file with "File > Save to HDF5 file…", or the button in the toolbar.

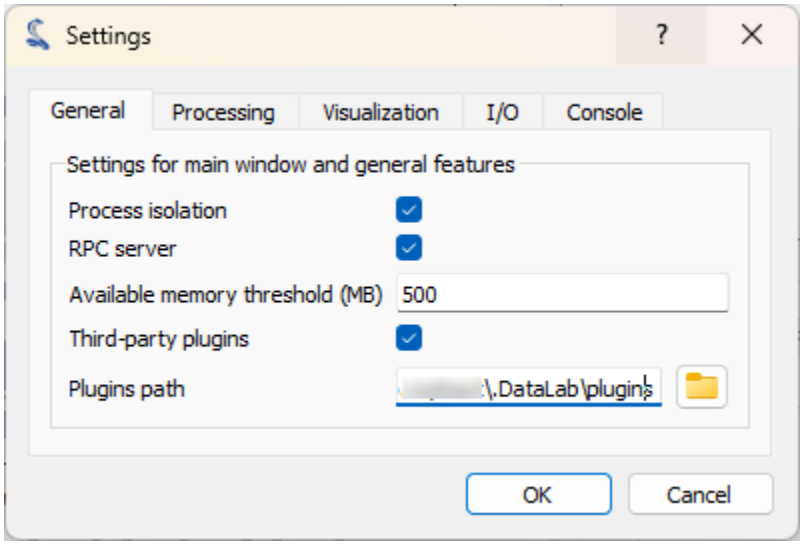


Fig. 24: In the "General" tab, we can see the "Plugins path" field. This is the path where DataLab will look for plugins. We can add a new plugin by copying/pasting the plugin file in this directory.

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

See also:

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

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

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

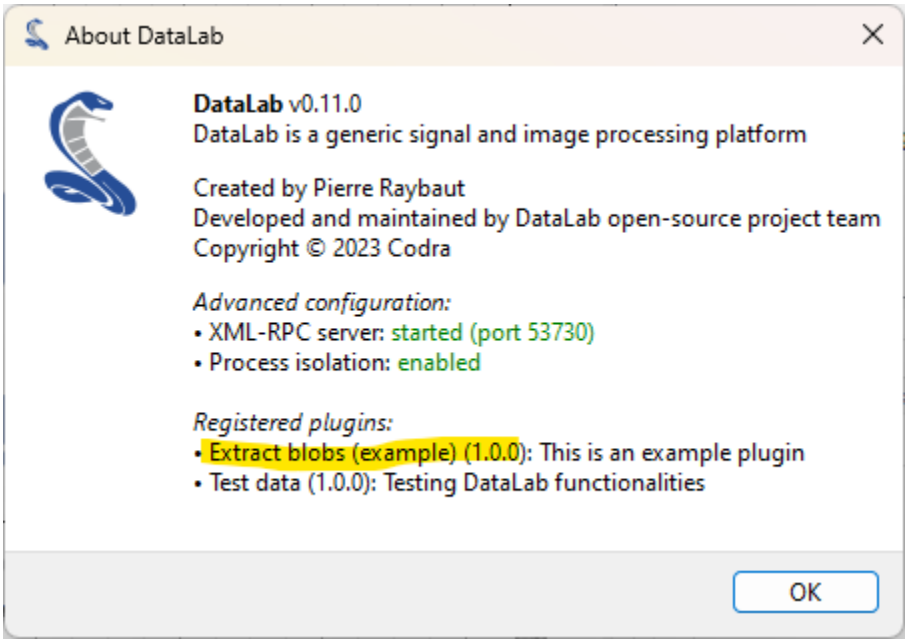


Fig. 25: The "About DataLab" dialog shows the list of available plugins.

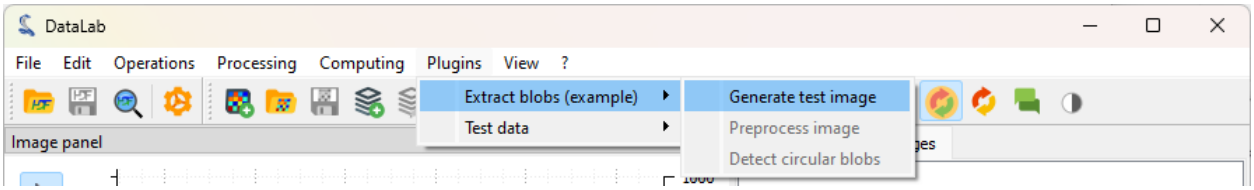


Fig. 26: Let's click on "Extract blobs (example) > Generate test image"

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

```
def generate_test_image(self) -> None:
    """Generate test image"""
    # Create a NumPy array:
    arr = np.random.normal(10000, 1000, (2048, 2048))
    for _ in range(10):
        row = np.random.randint(0, arr.shape[0])
        col = np.random.randint(0, arr.shape[1])
        rr, cc = skimage.draw.disk((row, col), 40, shape=arr.shape)
        arr[rr, cc] -= np.random.randint(5000, 6000)
    icenter = arr.shape[0] // 2
```

(continues on next page)

(continued from previous page)

```
    rr, cc = skimage.draw.disk((icenter, icenter), 200, shape=arr.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
    image = cdl.obj.create_image("Test image", data, units=("mm", "mm", "lsb"))
    self.proxy.add_object(image)
```

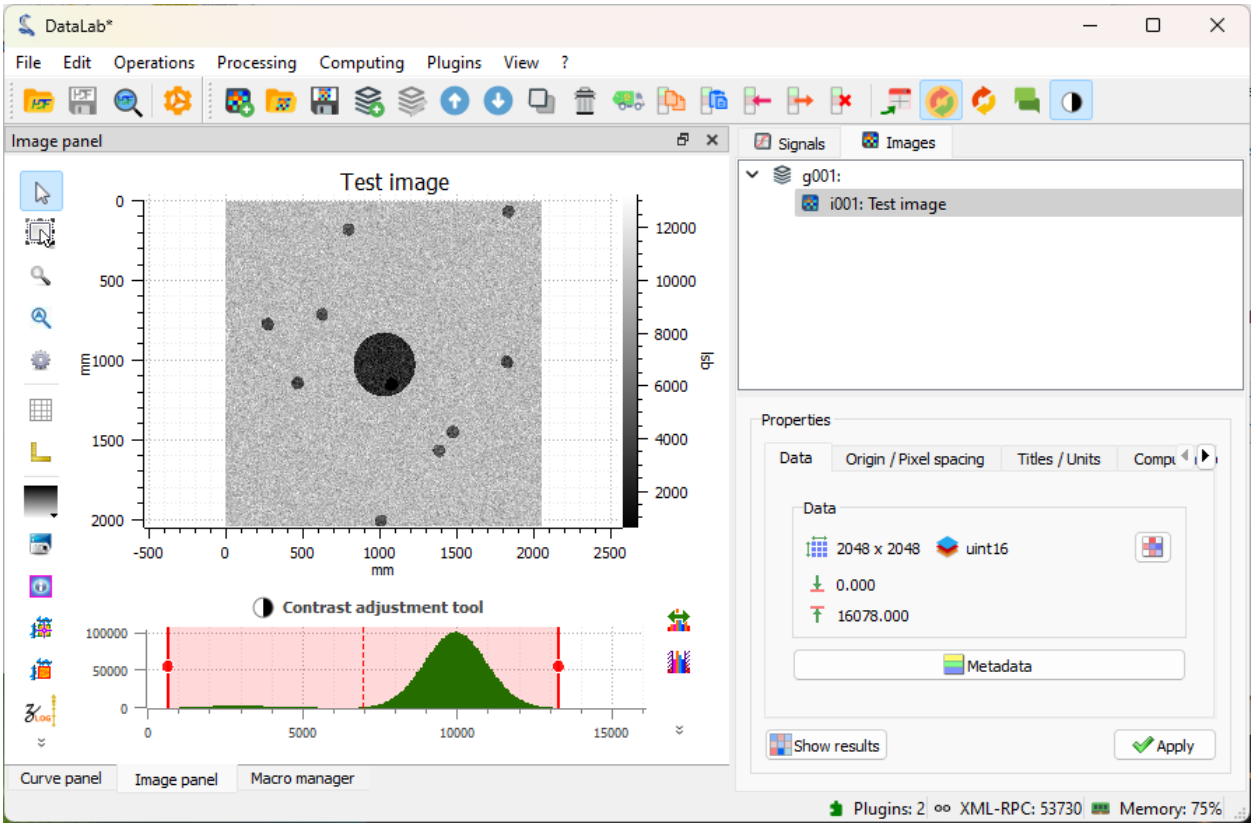


Fig. 27: The plugin has generated a test image, and added it to the "Images" panel. The image shows a few blobs, with a central dark disk, and a noisy background.

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 use the same DataLab native features as the plugin, manually.

The image is a bit noisy, and also quite large. Let's reduce the size of the image while denoising it a bit by binning the image by a factor of 2.

Let's apply a moving median filter to the image, to denoise it a bit more.

Now, let's detect the blobs on the image.

Note: If you want to show the computing results again, you can select the "Show results" entry in the "Computing" menu, or the "Show results" button, below the image list:

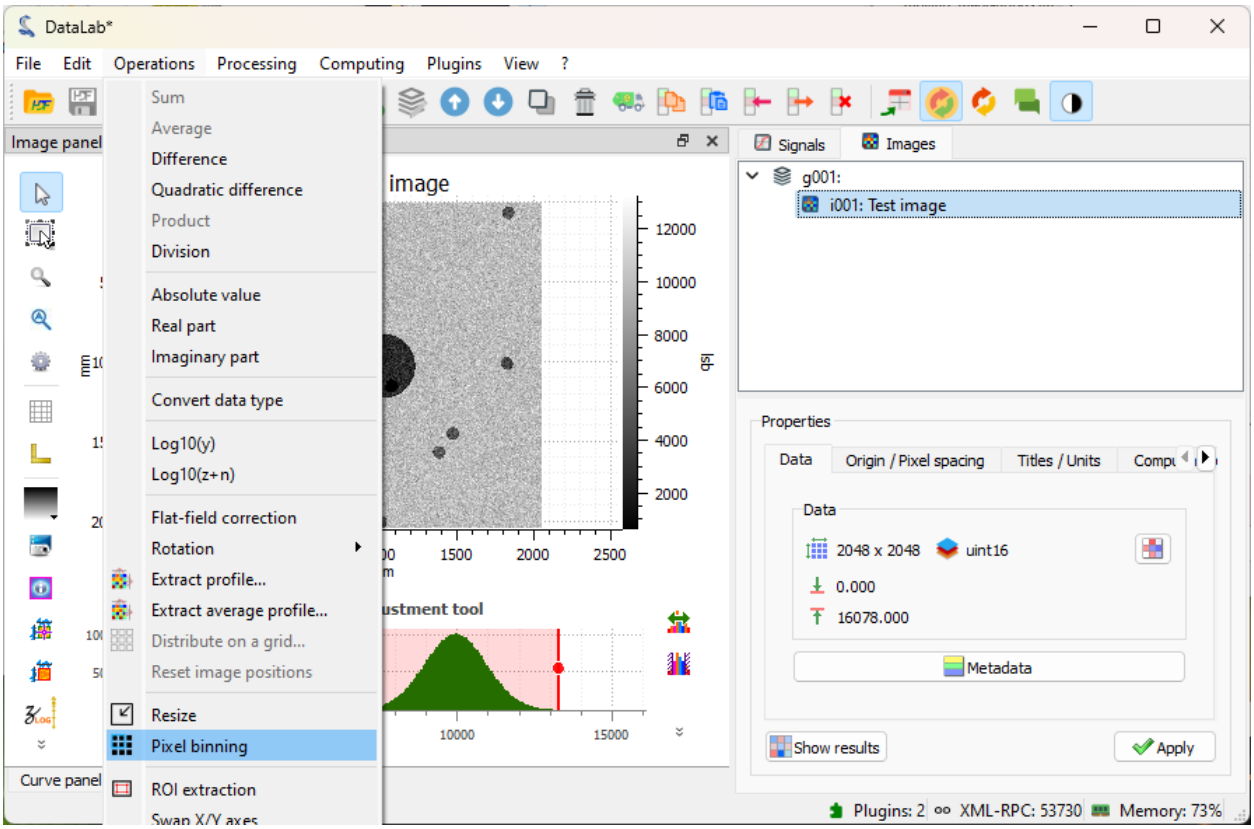


Fig. 28: Click on “Operations > Pixel binning”.

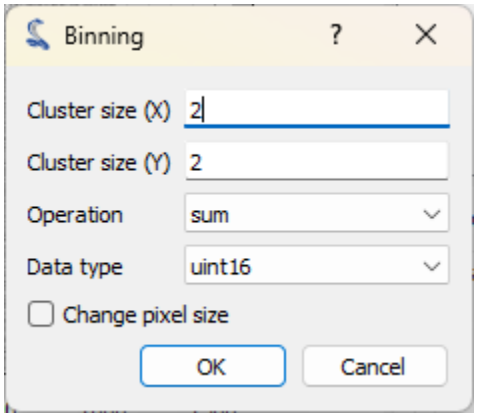


Fig. 29: The “Binning” dialog opens. Set the binning factor to 2, and click on “OK”.

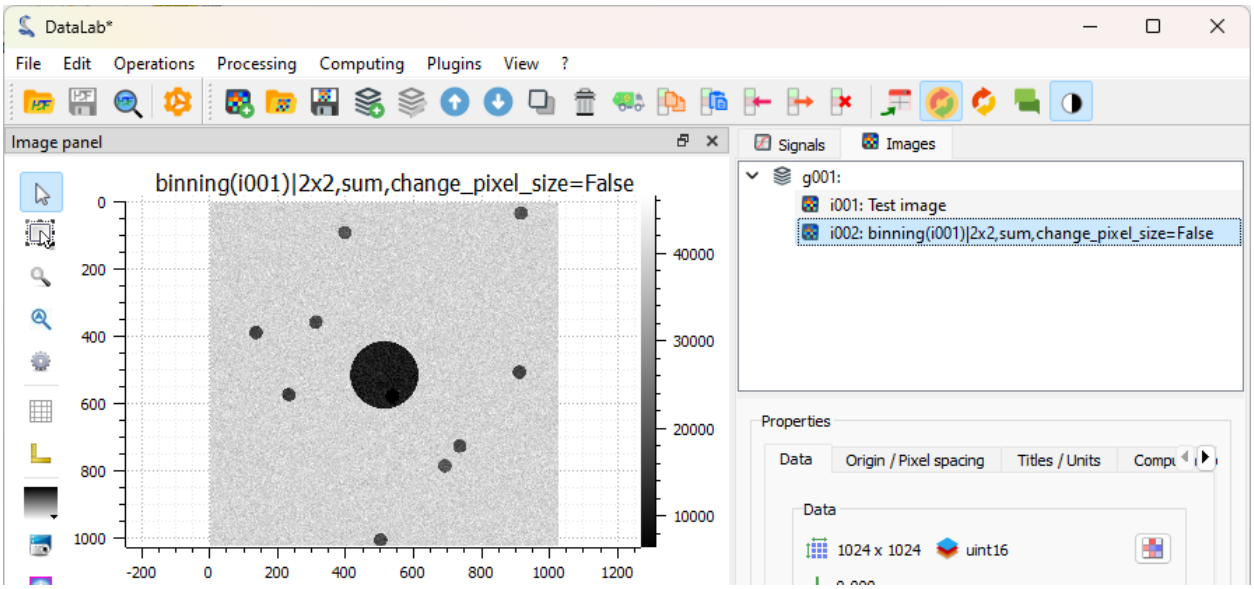


Fig. 30: The binned image is added to the "Images" panel. It is now easier to see the blobs (even if they were already quite visible on the original image: this is just an example), and the image will be faster to process.

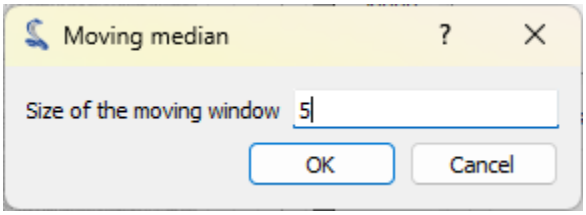


Fig. 31: Click on "Processing > Moving median" entry, and set the window size to 5.

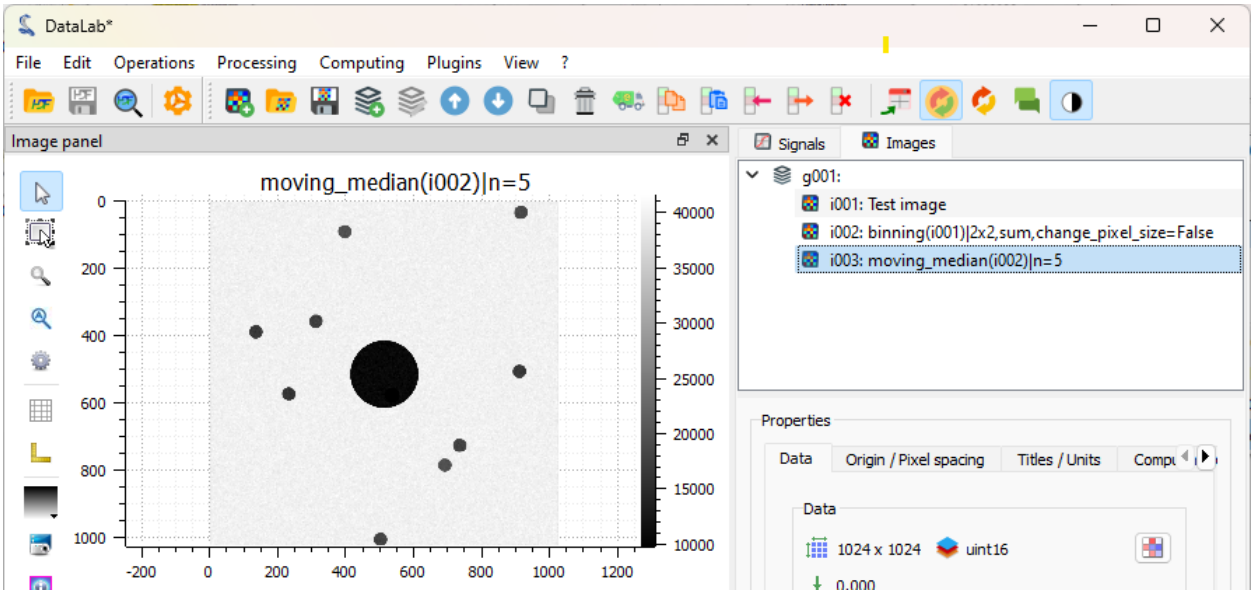


Fig. 32: The filtered image is added to the "Images" panel. Denoising is quite efficient.

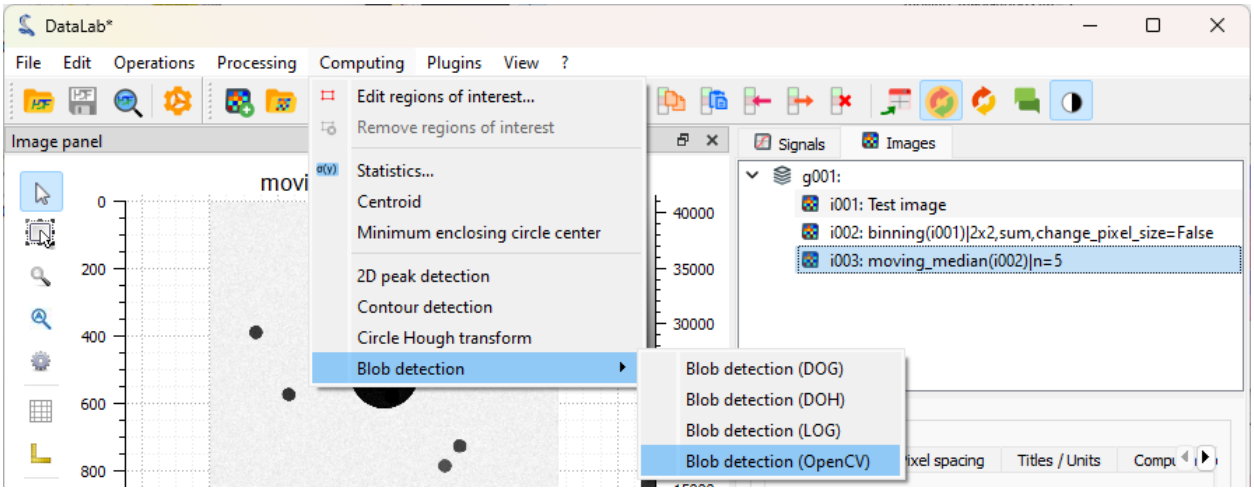


Fig. 33: Click on “Computing > Blob detection > Blob detection (OpenCV)”.

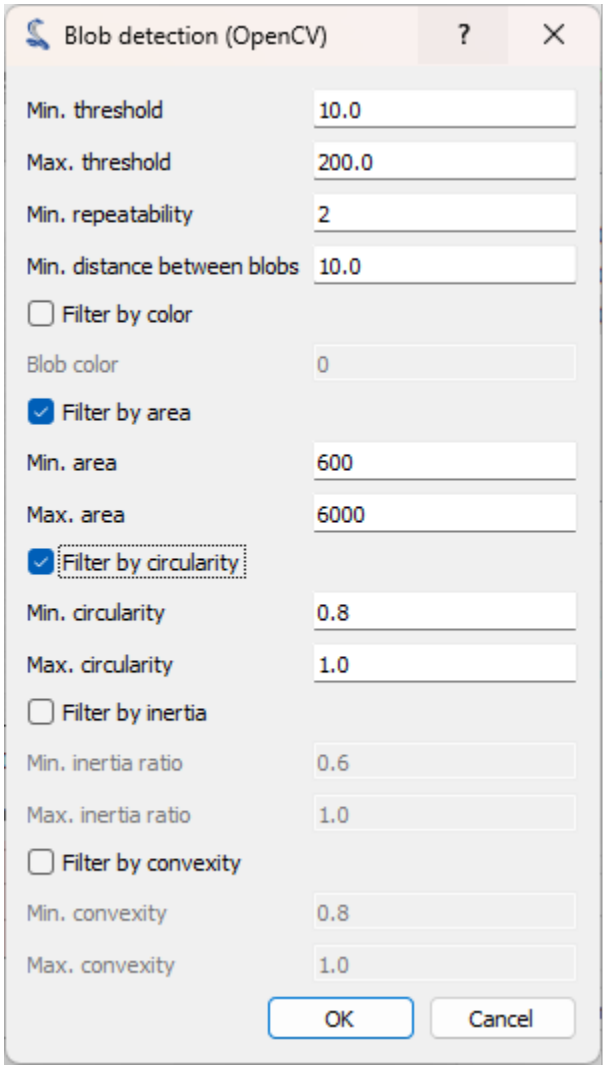


Fig. 34: The “Blob detection (OpenCV)” dialog opens. Set the parameters as shown on the screenshot, and click on “OK”.

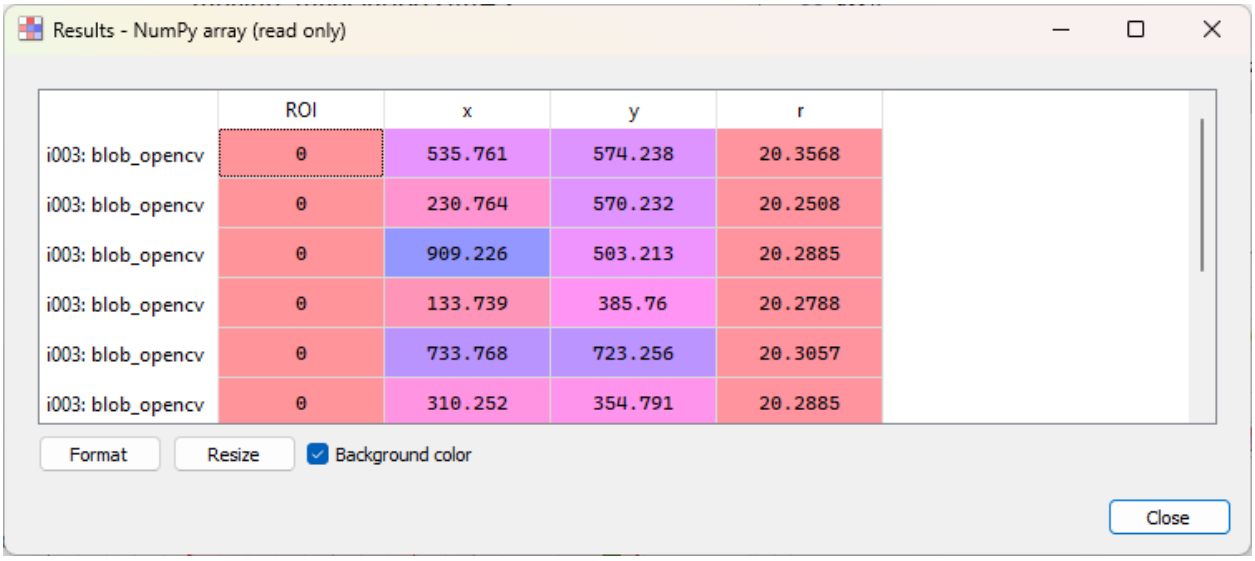
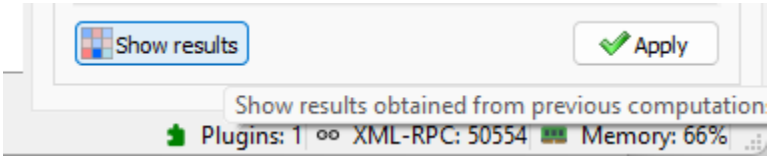


Fig. 35: The "Results" dialog opens, showing the detected blobs: one line per blob, with the blob coordinates and radius.



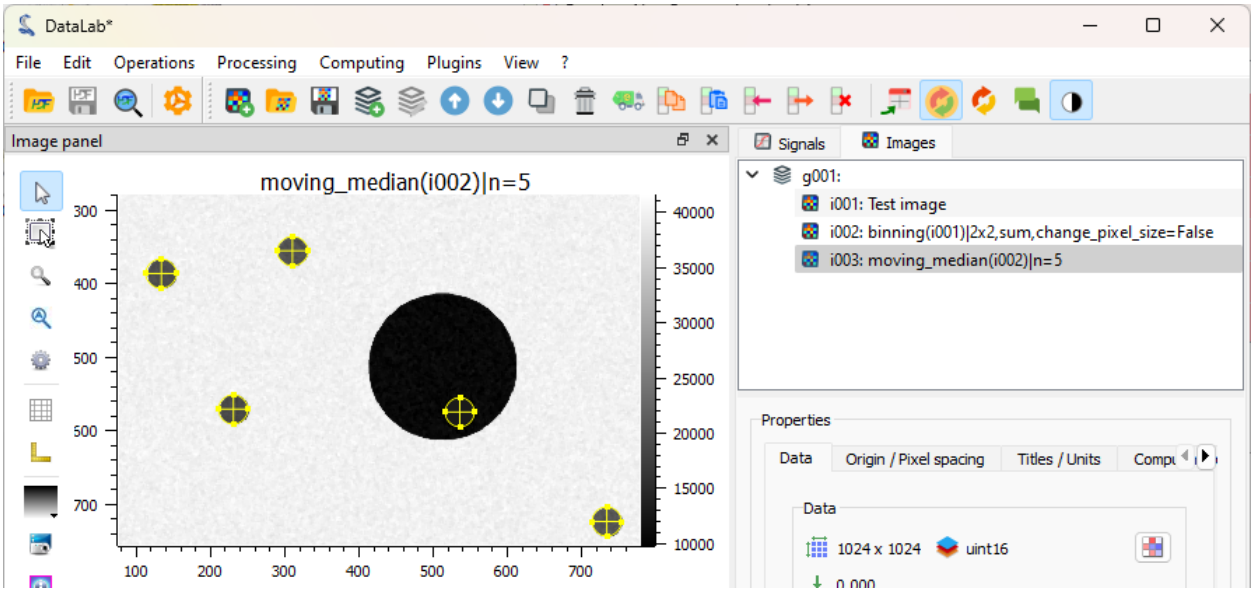


Fig. 36: The detected blobs are also added to the image metadata, and can be seen in the visualization panel on the left.

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

If you want to load the workspace again, you can use the "File > Open HDF5 file…" (or the button in the toolbar) to

load the whole workspace, or the "File > Browse HDF5 file…" (or the button in the toolbar) to load only a selection of data sets from the workspace.

Measuring Fabry-Perot fringes

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

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

First, we open DataLab and load the images:

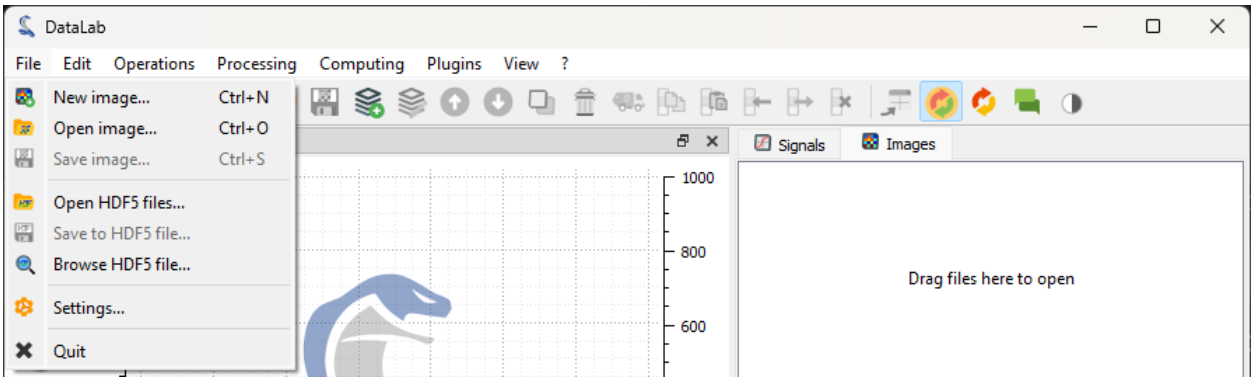


Fig. 37: Open the image files with "File > Open…", or with the button in the toolbar, or by dragging and dropping the files into DataLab (on the panel on the right).

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.

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

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

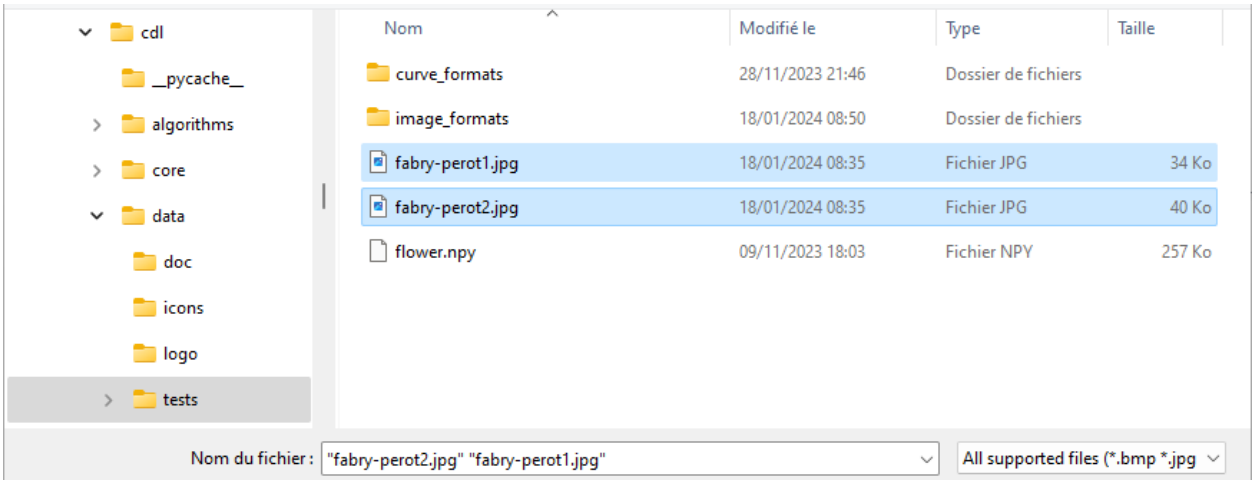


Fig. 38: Select the test images “fabry_perot1.jpg” and “fabry_perot2.jpg” and click “Open”.

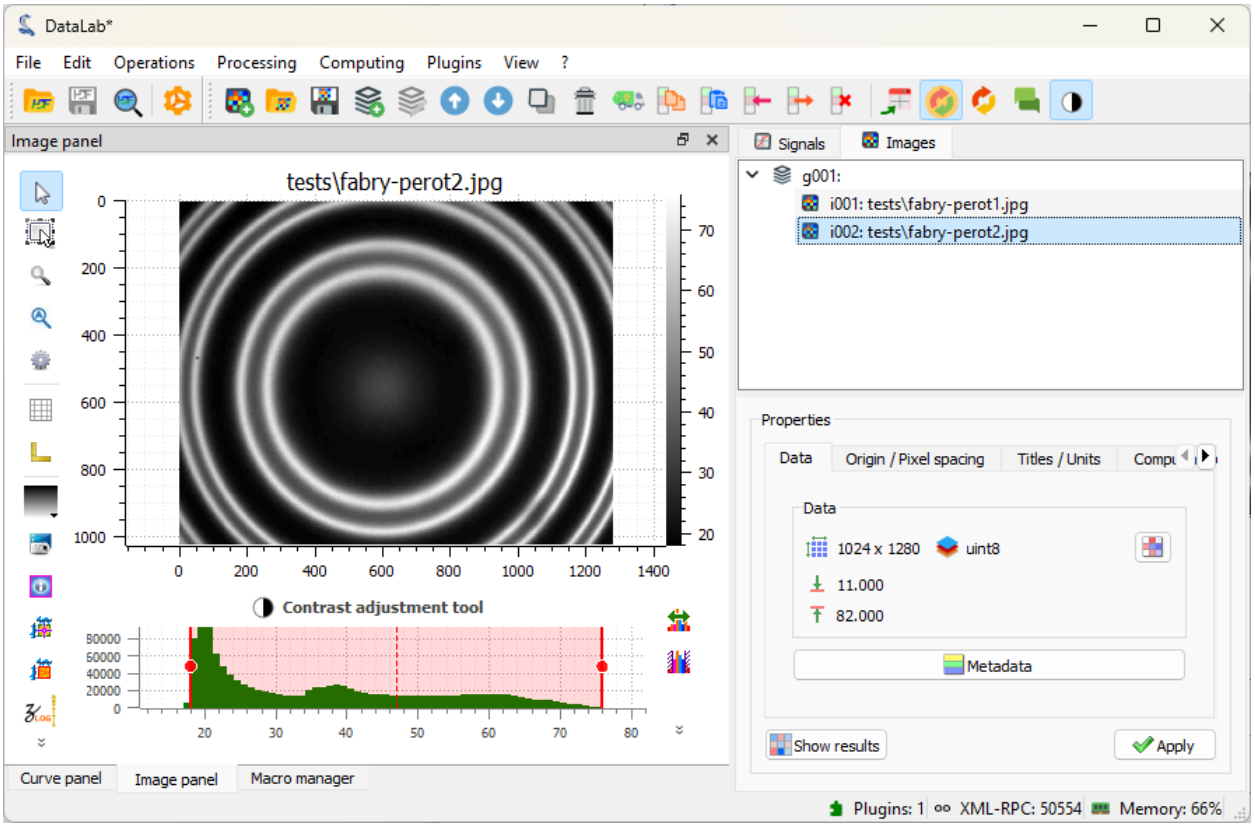


Fig. 39: Zoom in and out with the right mouse button. Pan the image with the middle mouse button.

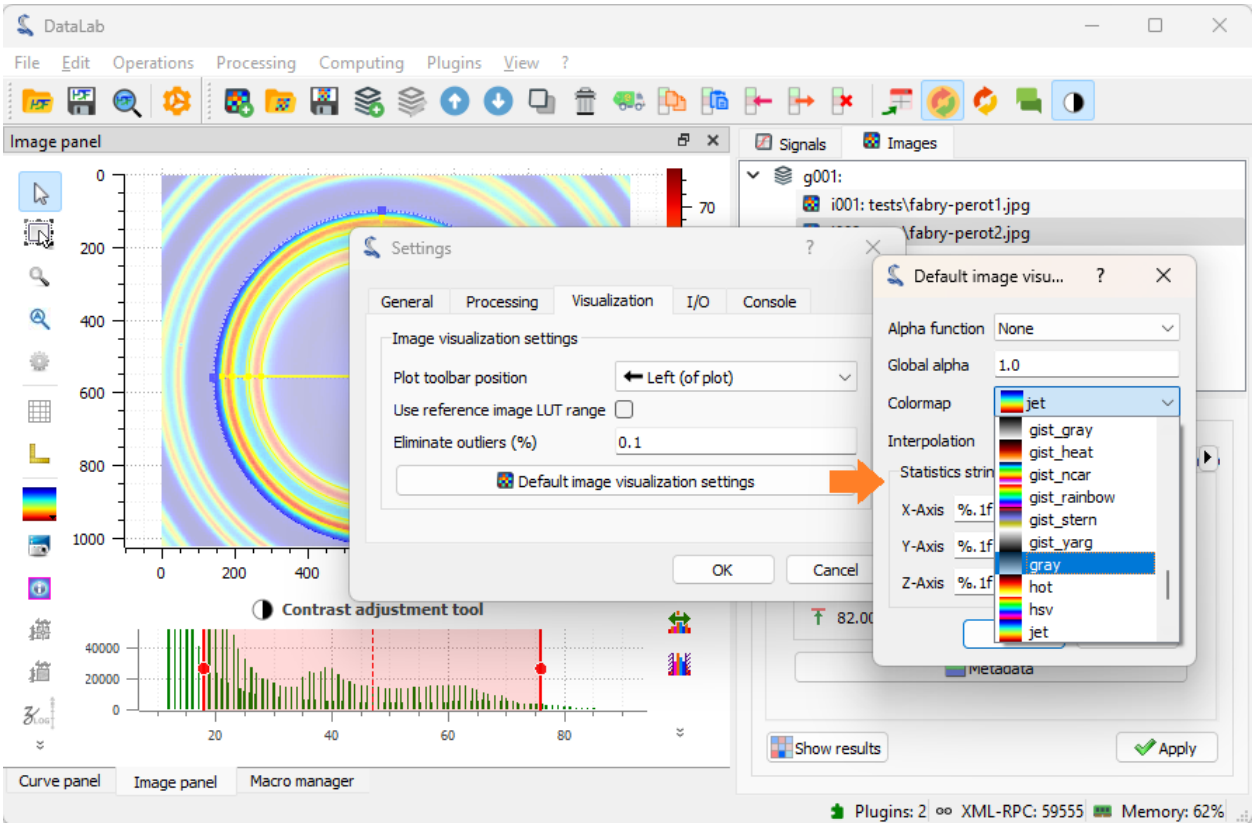


Fig. 40: Select the “Visualization” tab, and select the “gray” colormap.

Then, let’s define a circular region of interest (ROI) around the central fringe.

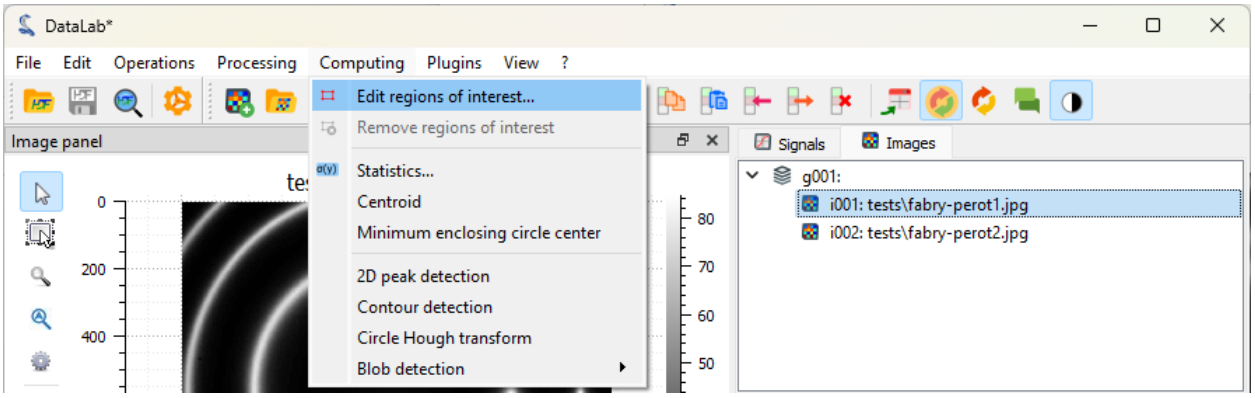


Fig. 41: Select the “Edit regions of interest” tool in the “Computing” menu.

Now, let’s detect the contours in the ROI and fit them to circles.

Note: If you want to show the computing results again, you can select the “Show results” entry in the “Computing” menu, or the “Show results” button, below the image list:

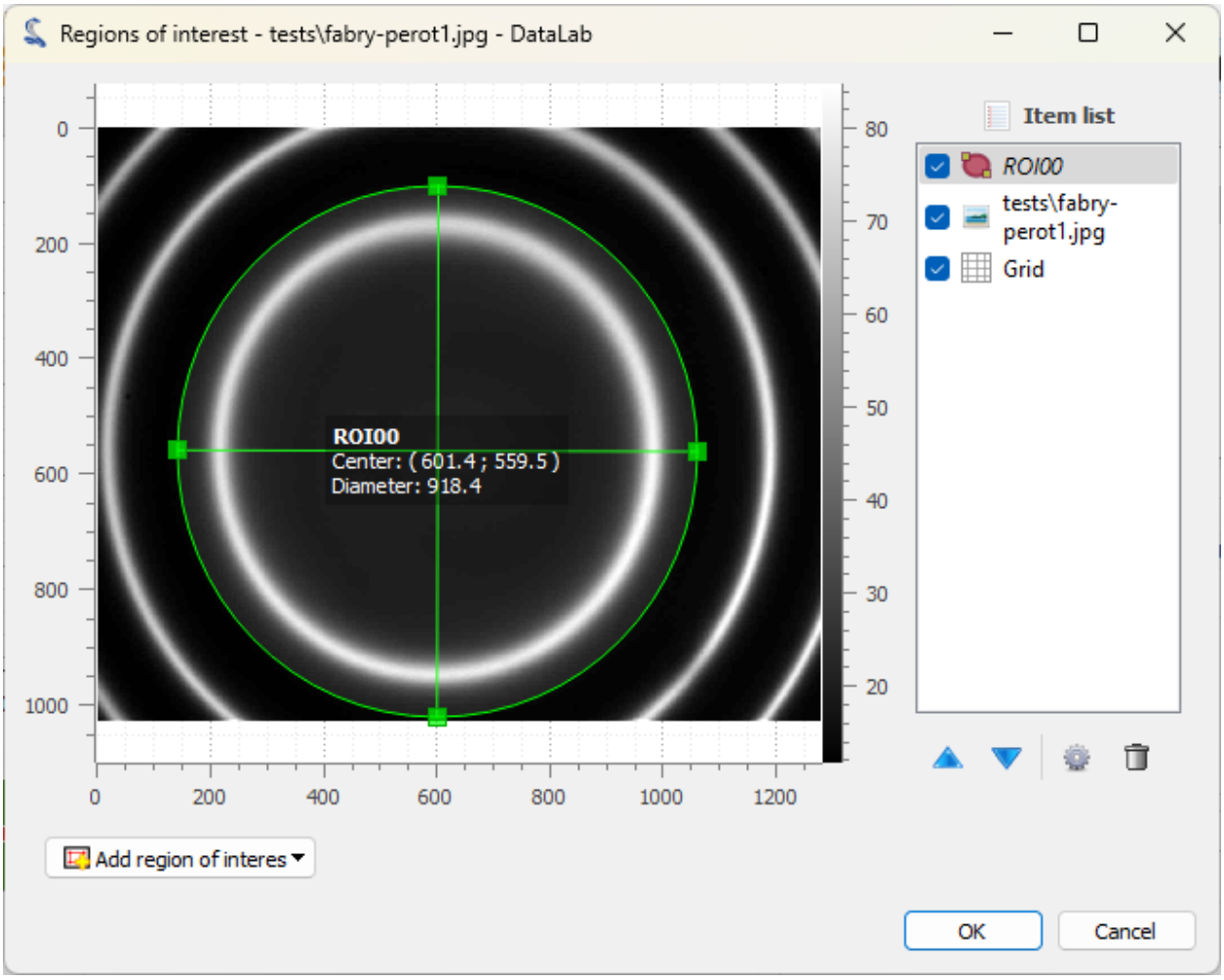


Fig. 42: The "Regions of interest" dialog opens. Click "Add region of interest" and select a circular ROI. Resize the predefined ROI by dragging the handles. Note that you may change the ROI radius while keeping its center fixed by pressing the "Ctrl" key. Click "OK" to close the dialog.

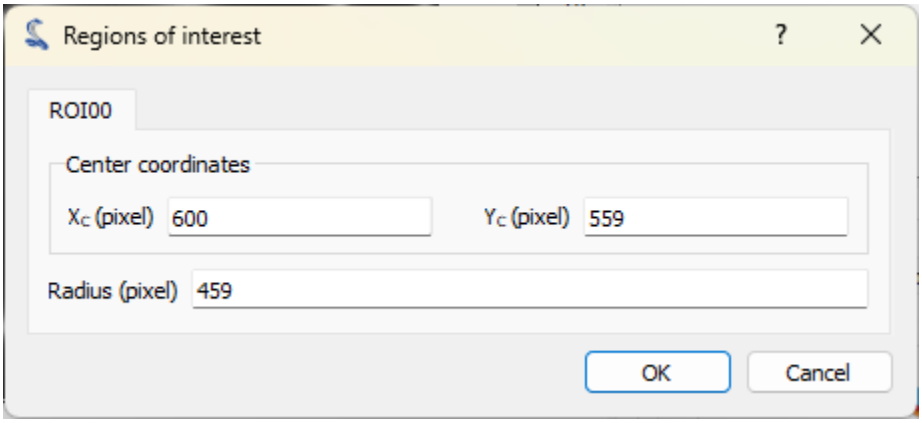


Fig. 43: Another dialog box opens, and asks you to confirm the ROI parameters. Click "OK".

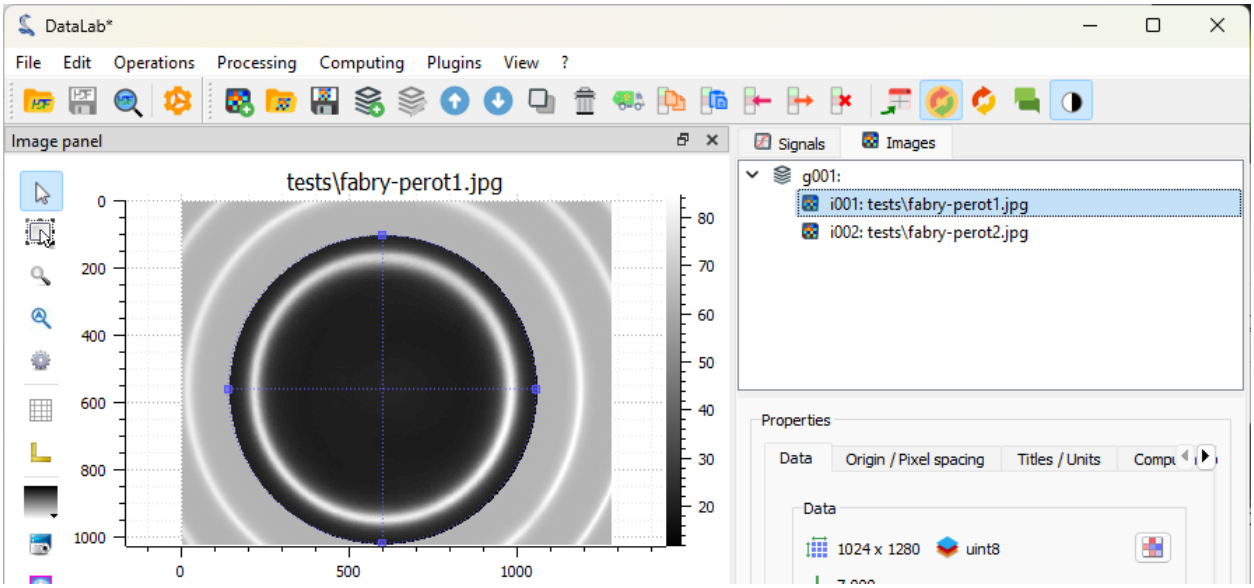


Fig. 44: The ROI is displayed on the image: masked pixels are grayed out, and the ROI boundary is displayed in blue (note that, internally, the ROI is defined by a binary mask, i.e. image data is represented as a NumPy masked array).

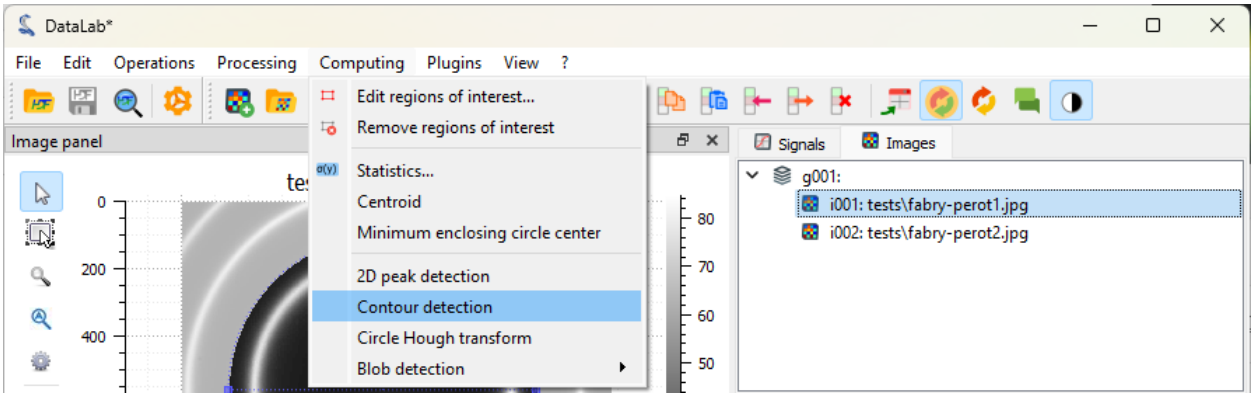


Fig. 45: Select the "Contour detection" tool in the "Computing" menu.

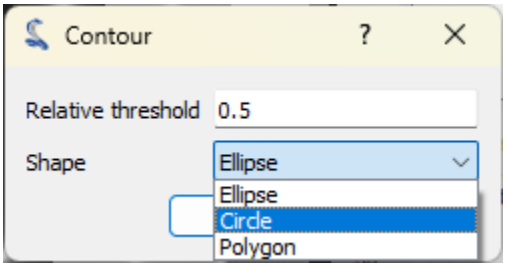


Fig. 46: The "Contour" parameters dialog opens. Select the shape "Circle" and click "OK".

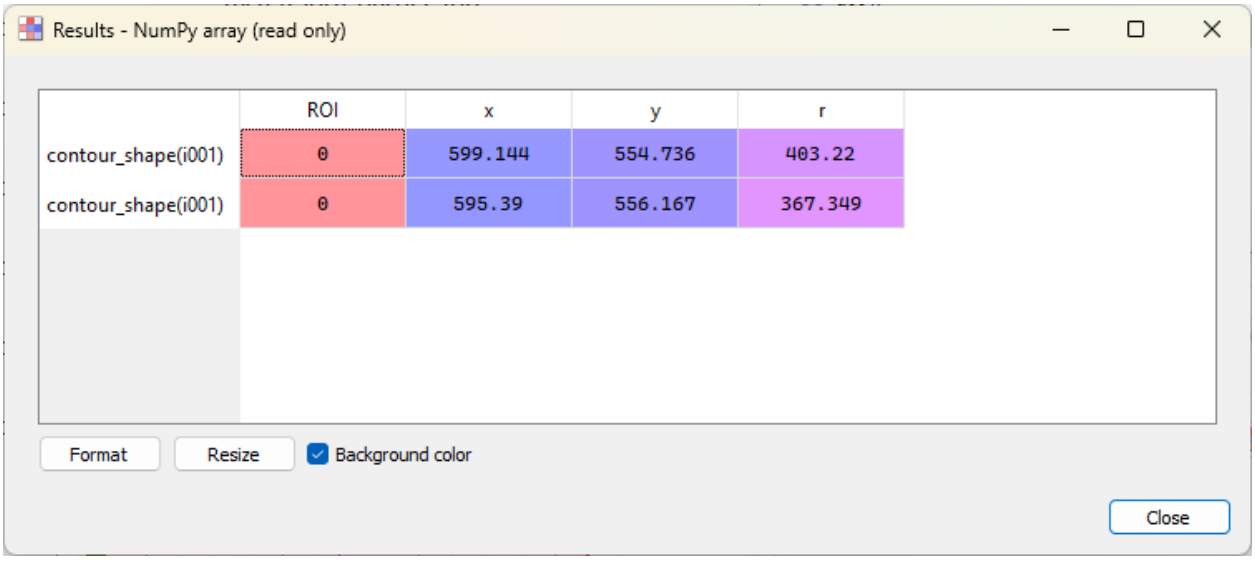


Fig. 47: The “Results” dialog opens, and displays the fitted circle parameters. Click “OK”.

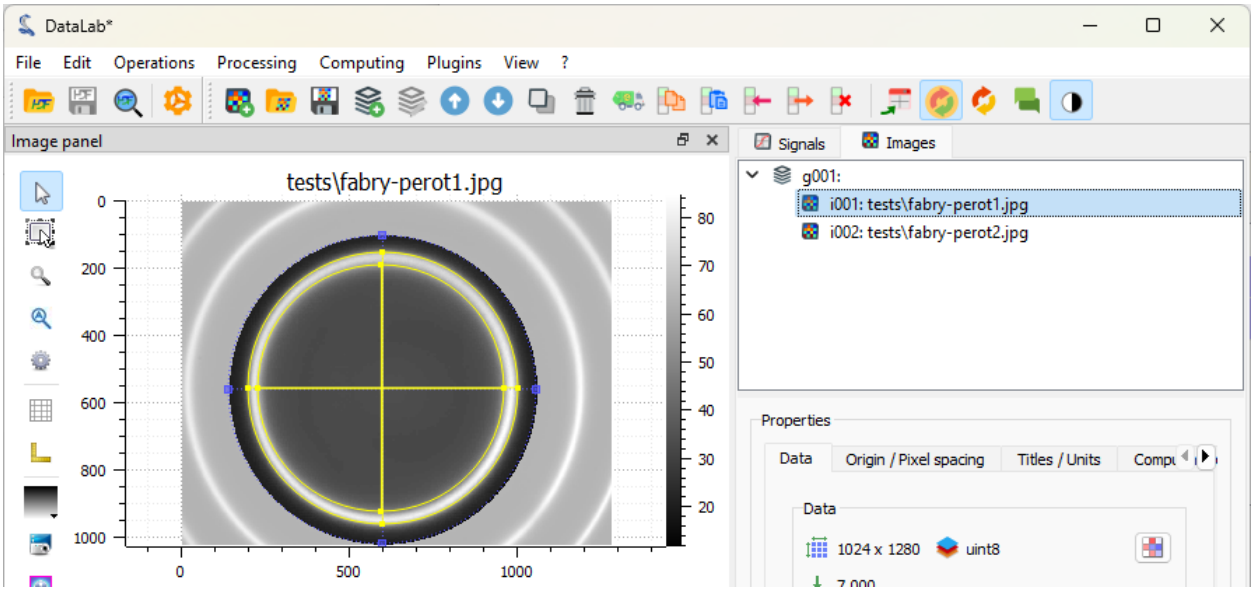
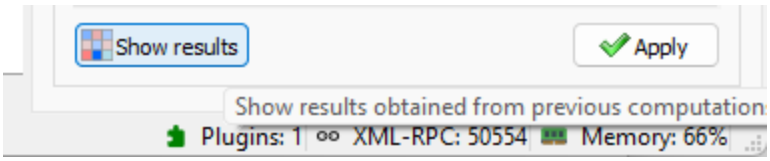


Fig. 48: The fitted circles are displayed on the image.



The images (or signals) can also be displayed in a separate window, by clicking on the "View in a new window" entry in the "View" menu (or the button in the toolbar). This is useful to compare side by side images or signals.

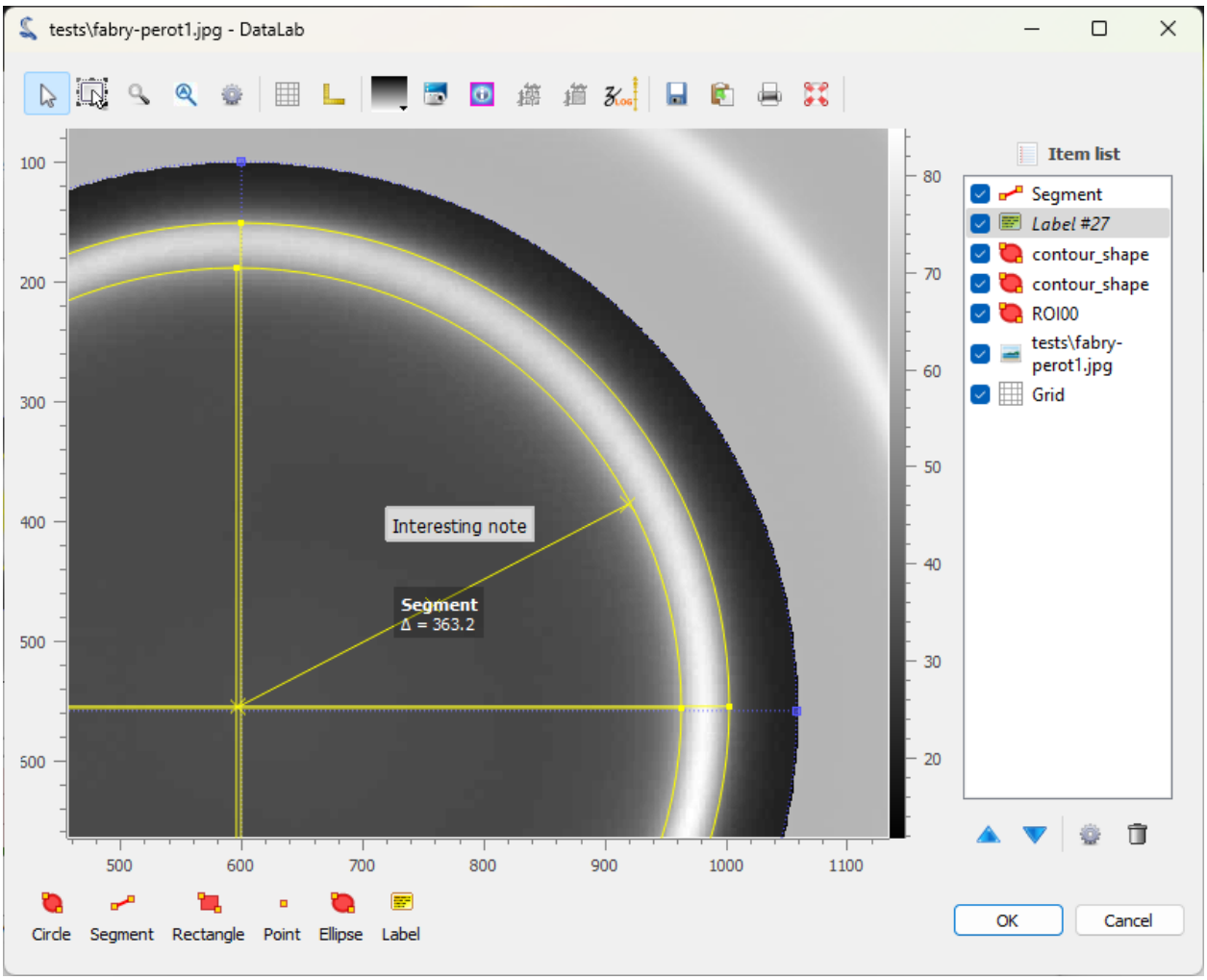


Fig. 49: The image is displayed in a separate window. The ROI and the fitted circles are also displayed. Annotations can be added to the image by clicking on the buttons at the bottom of the window. The annotations are stored in the metadata of the image, and together with the image data when the workspace is saved. Click on "OK" to close the window.

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

Now, let's delete the image metadata (including the annotations) to clean up the image.

If we want to define the exact same ROI on the second image, we can copy/paste the ROI from the first image to the second image, using the metadata.

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

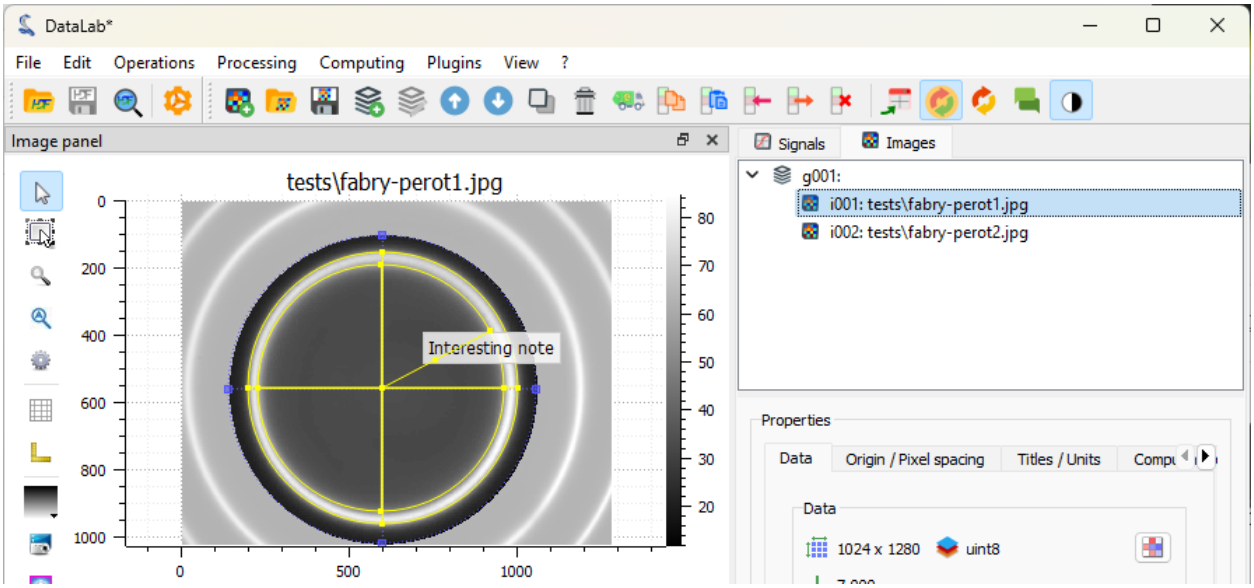


Fig. 50: The image is displayed in the main window, together with the annotations.

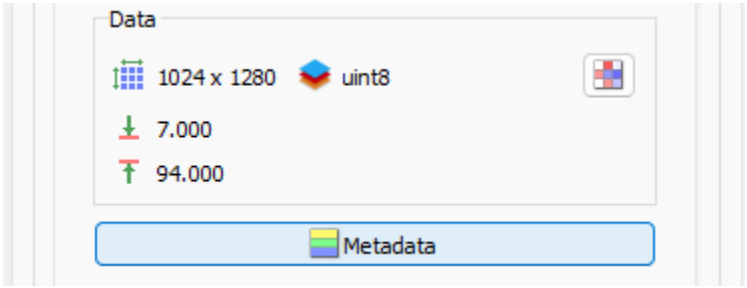


Fig. 51: The “Metadata” button is located below the image list.

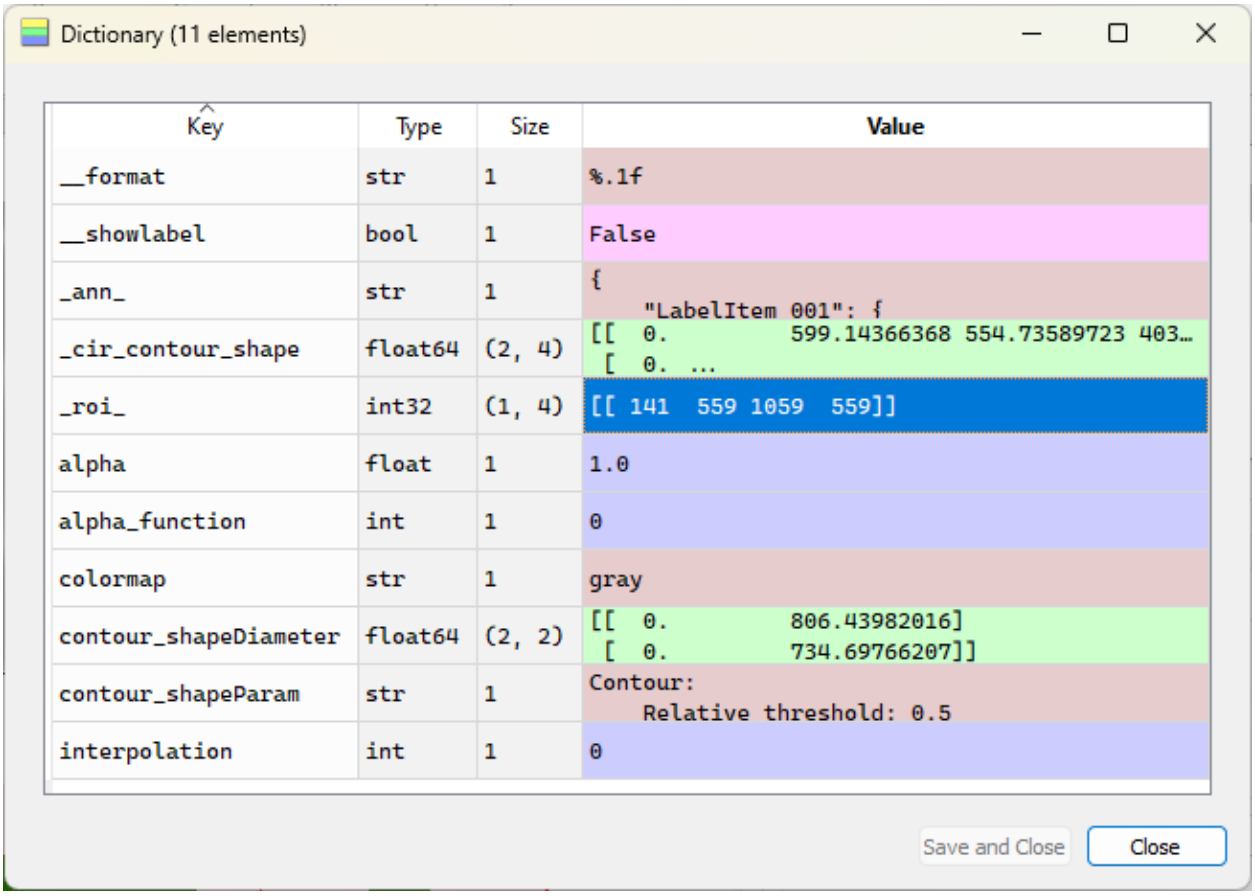


Fig. 52: The "Metadata" dialog opens. Among other information, it displays the annotations (in a JSON format), some style information (e.g. the colormap), and the ROI.

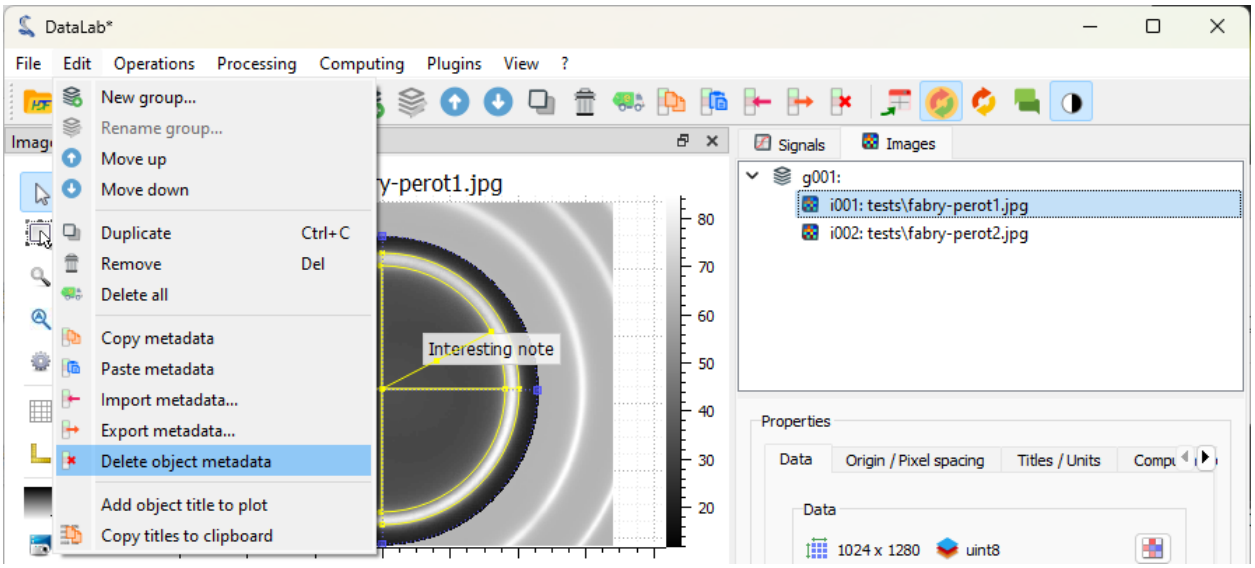


Fig. 53: Select the "Delete metadata" entry in the "Edit" menu, or the button in the toolbar.

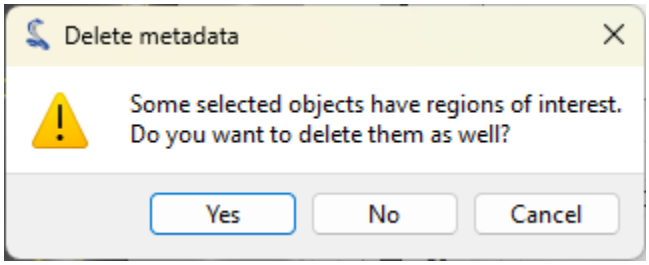


Fig. 54: The “Delete metadata” dialog opens. Click “No” to keep the ROI and delete the rest of the metadata.

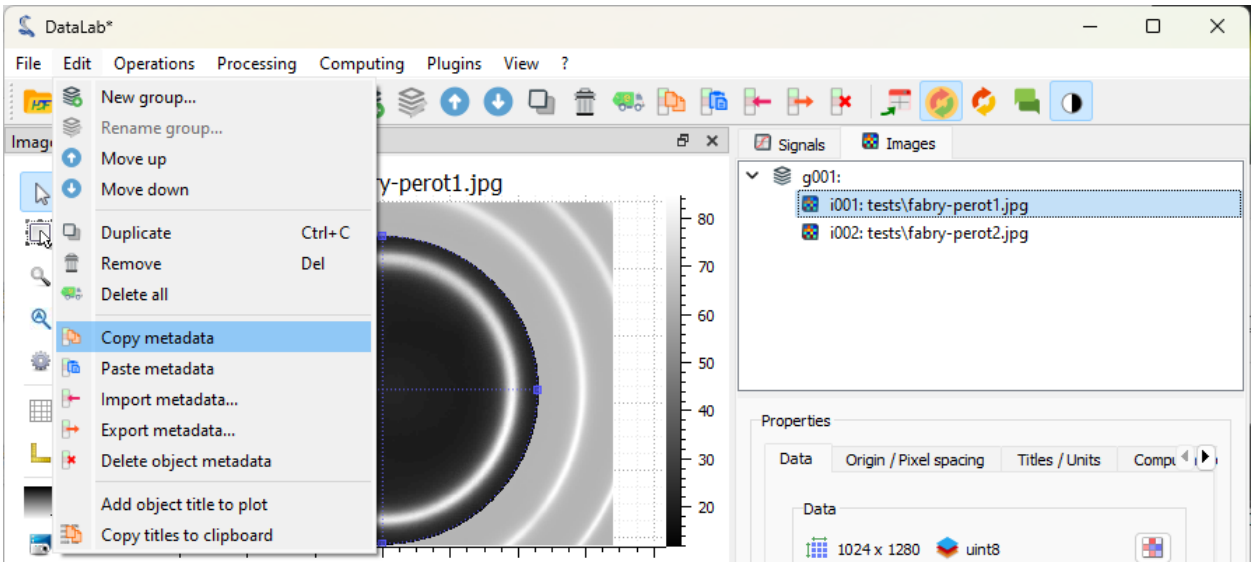


Fig. 55: Select the “Copy metadata” entry in the “Edit” menu, or the button in the toolbar.

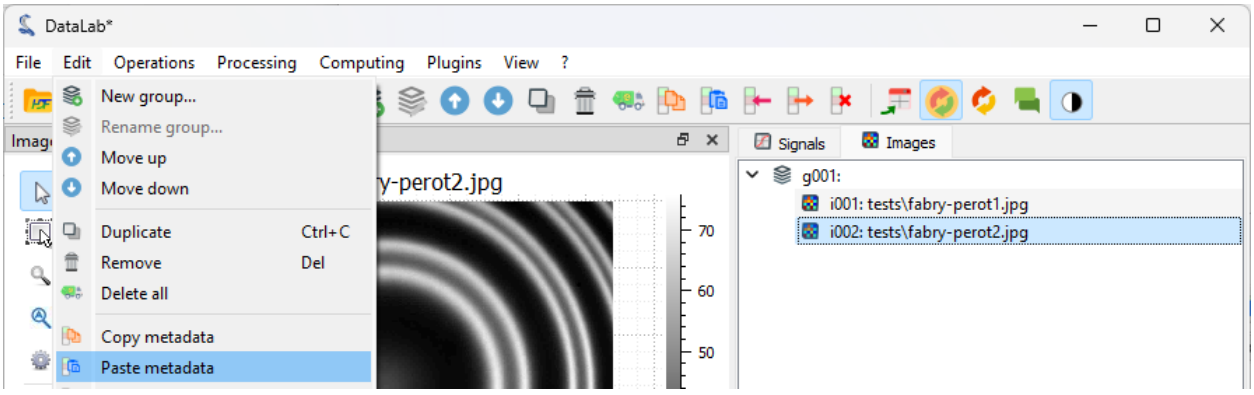


Fig. 56: Select the second image in the “Images” panel, then select the “Paste metadata” entry in the “Edit” menu, or the button in the toolbar.

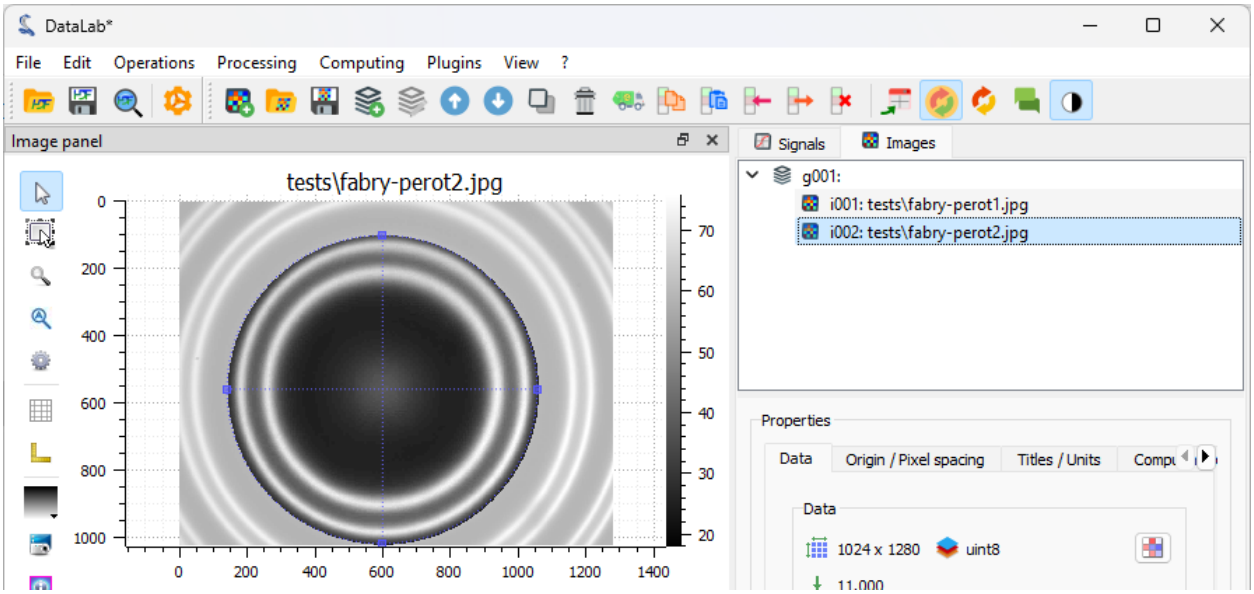


Fig. 57: The ROI is added to the second image.

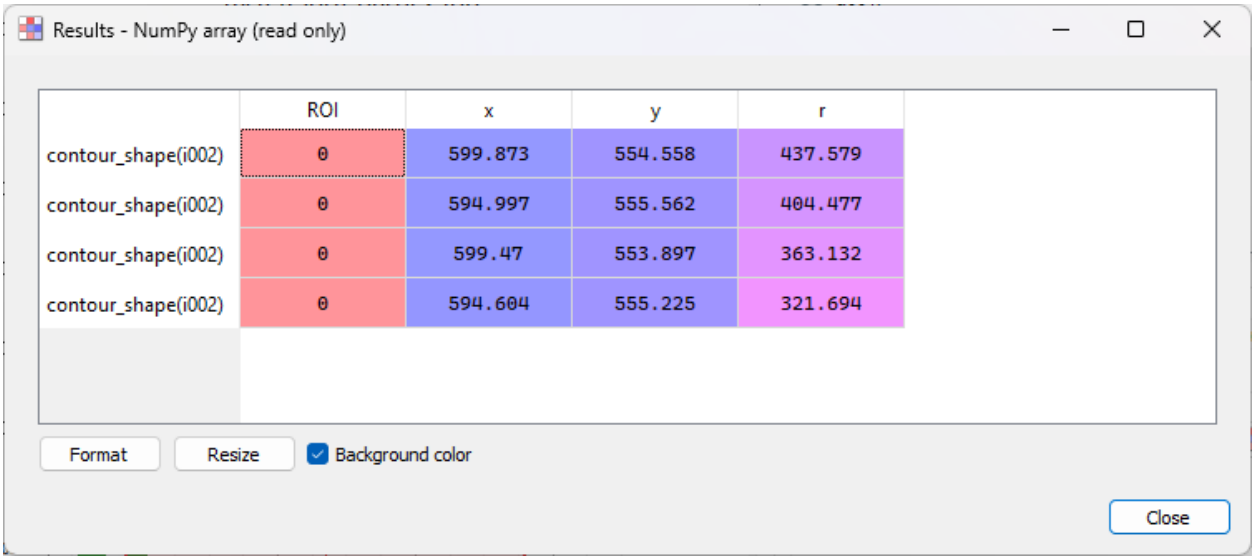


Fig. 58: Select the “Contour detection” tool in the “Computing” menu, with the same parameters as before (shape “Circle”). On this image, there are two fringes, so four circles are fitted. The “Results” dialog opens, and displays the fitted circle parameters. Click “OK”.

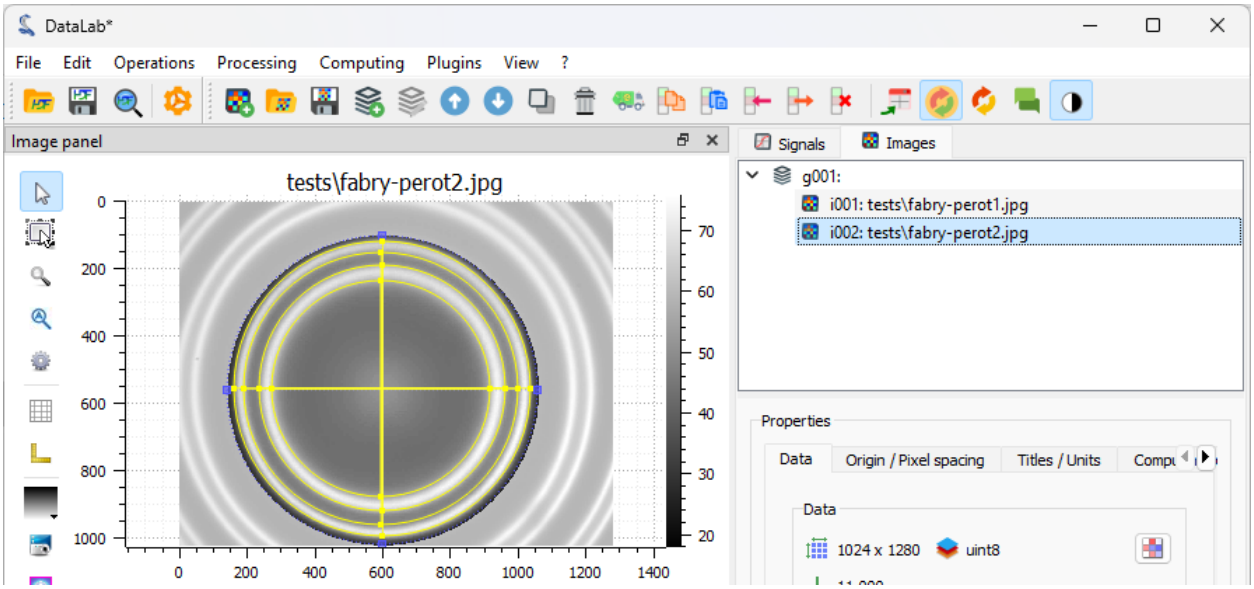


Fig. 59: The fitted circles are displayed on the image.

- Either select the "Line profile…" entry in the "Operations > 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, by selecting the "Line profile…" entry : that is the most straightforward way to extract a profile from an image, and it corresponds to the `compute_profile` method of DataLab's API (so it can be used in a script, a plugin or a macro).

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, by activating the "Cross section" tool in the vertical toolbar on the left of the visualization panel (this tool is a PlotPy feature). Before being able to use it, we need to select the image in the visualization panel (otherwise the tool is grayed out). Then, we can click on the image to display the intensity profile along the X axis. DataLab integrates a modified version of this tool, that allows to transfer the profile to the "Signals" panel for further processing.

Then, click on the "Process signal" button in the toolbar near the profile to transfer the profile to the "Signals" panel.

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 computing results, the visualization settings (colormaps, contrast, etc.), the metadata, and the annotations.

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

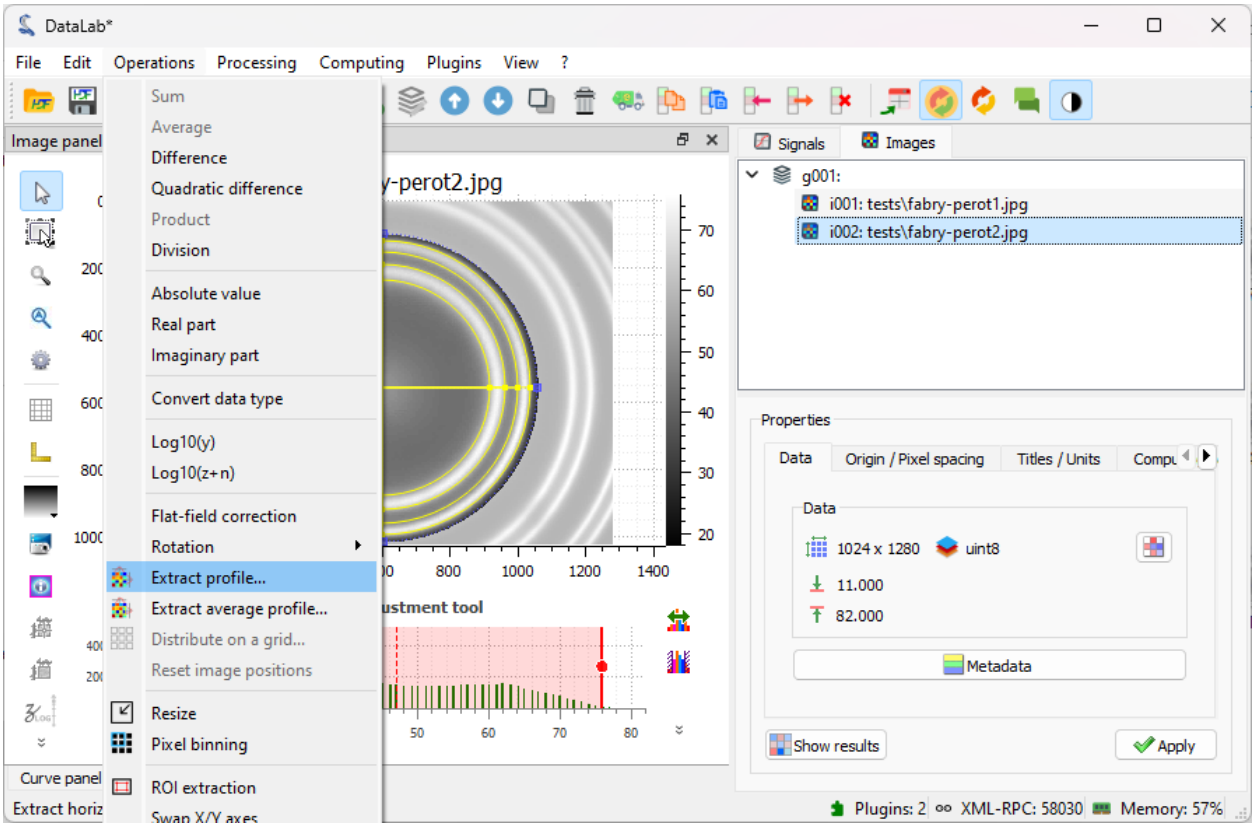


Fig. 60: Select the “Line profile…” entry in the “Operations” menu.

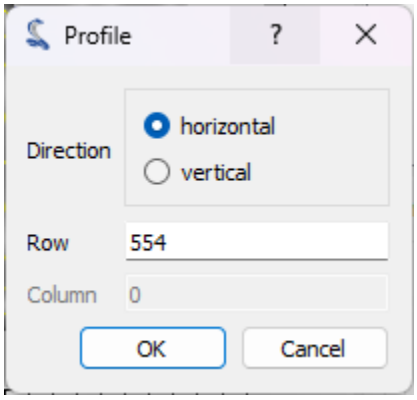


Fig. 61: The “Profile” dialog opens. Enter the row of the horizontal profile (or the column of the vertical profile) in the dialog box that opens. Click “OK”.

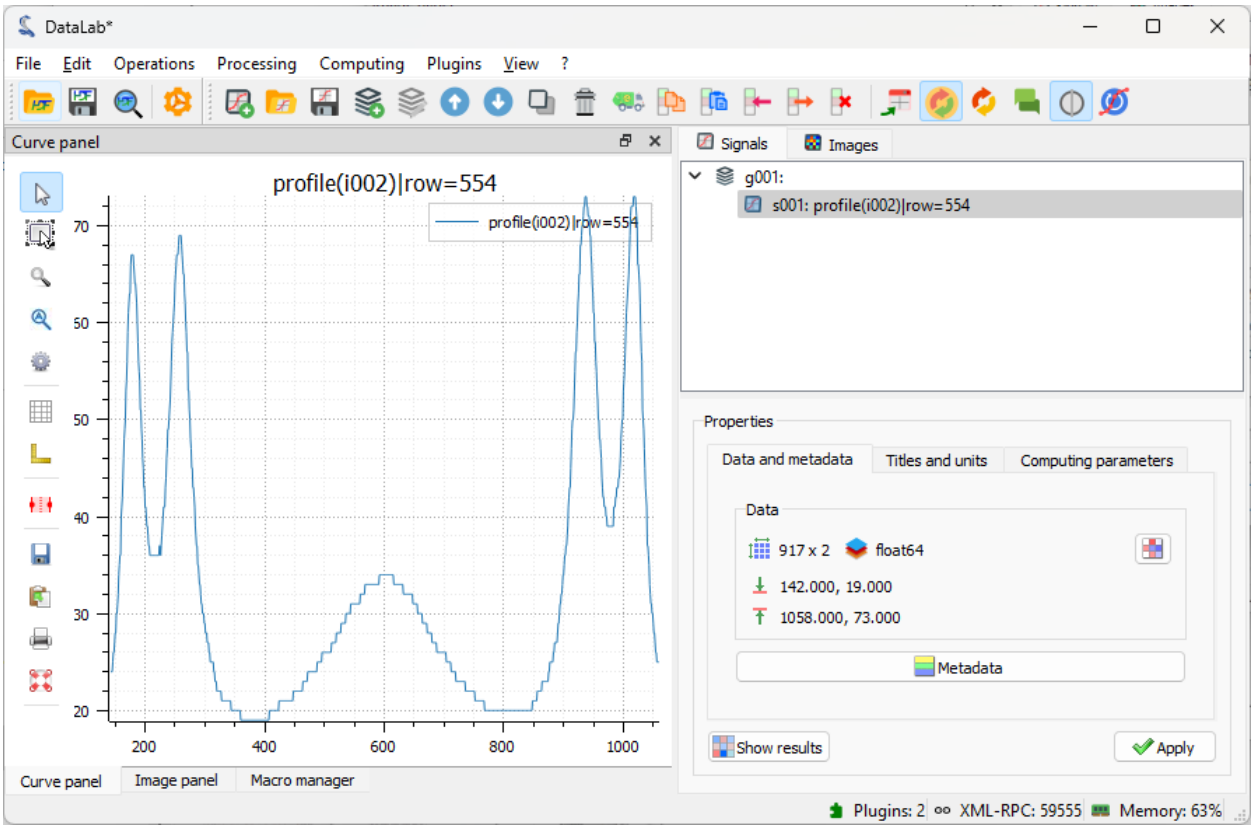


Fig. 62: The intensity profile is added to the "Signals" panel, and DataLab switches to this panel to display the profile.

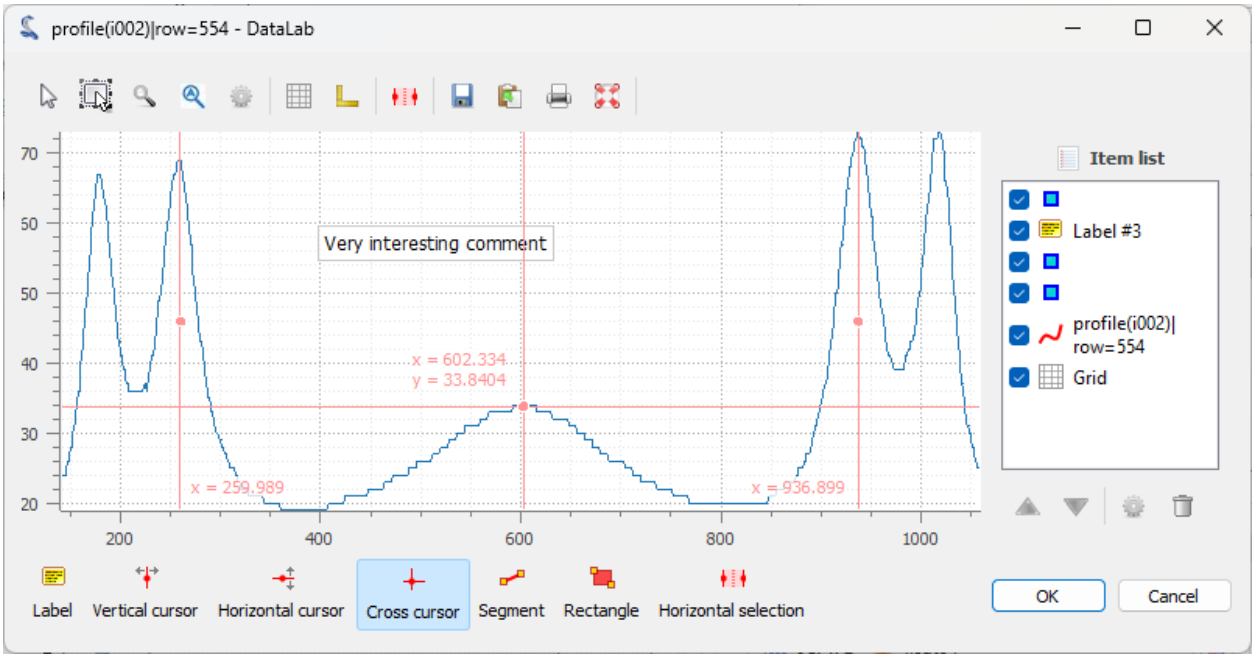


Fig. 63: The signal is displayed in a separate window. Here, we added vertical cursors and a very interesting text label. As for the images, the annotations are stored in the metadata of the signal, and together with the signal data when the workspace is saved. Click on "OK" to close the window.

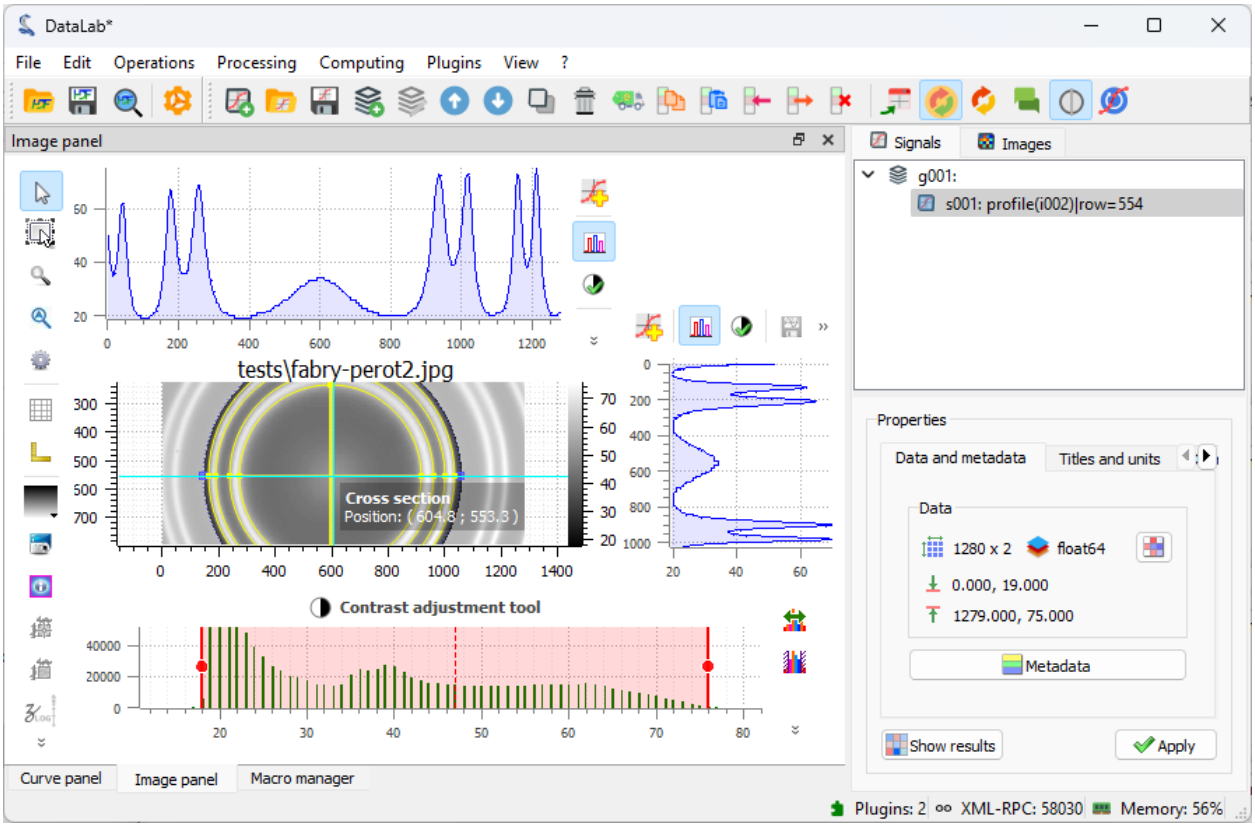


Fig. 64: Switch back to the "Images" panel, and select the image *in the visualization panel* (otherwise the "Cross section" tool is grayed out). Select the "Cross section" tool in the vertical toolbar, and click on the image to display the intensity profiles along the X and Y axes.

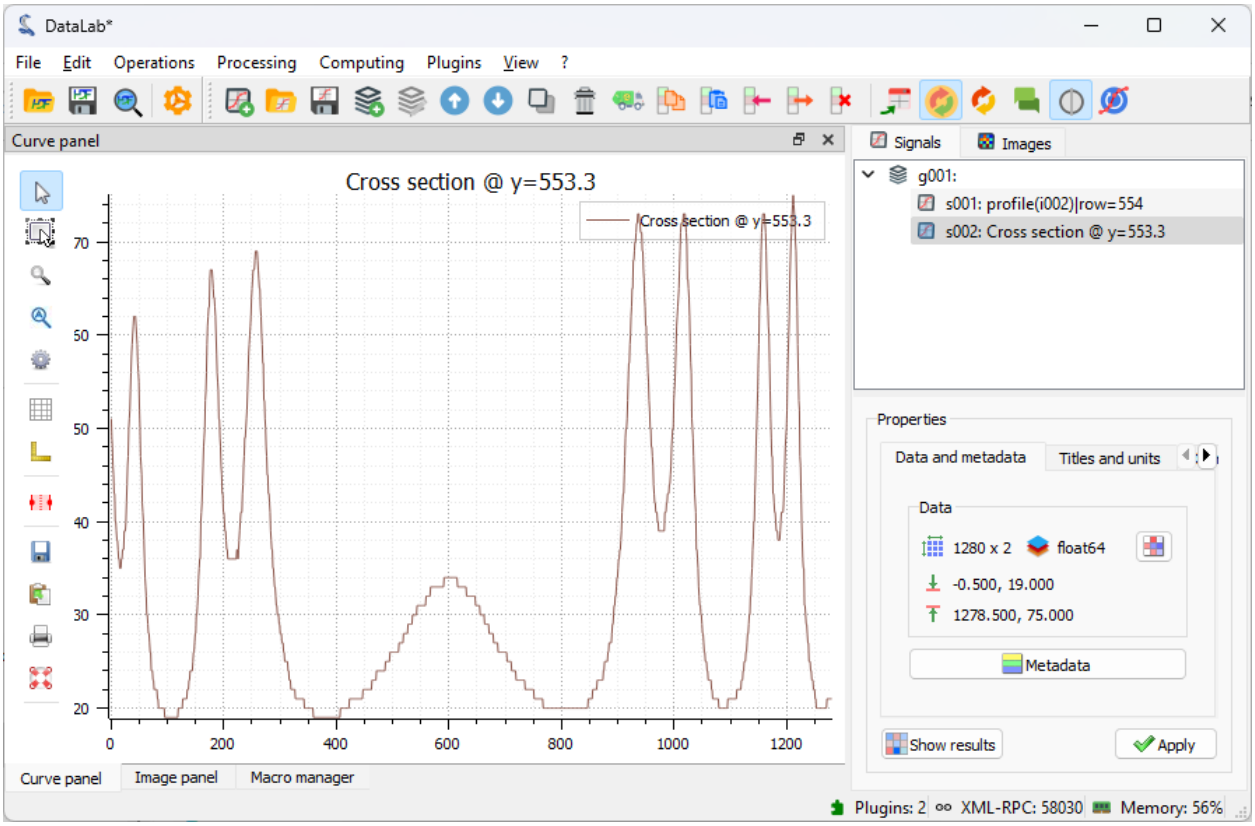


Fig. 65: The intensity profile is added to the "Signals" panel, and DataLab switches to this panel to display the profile.

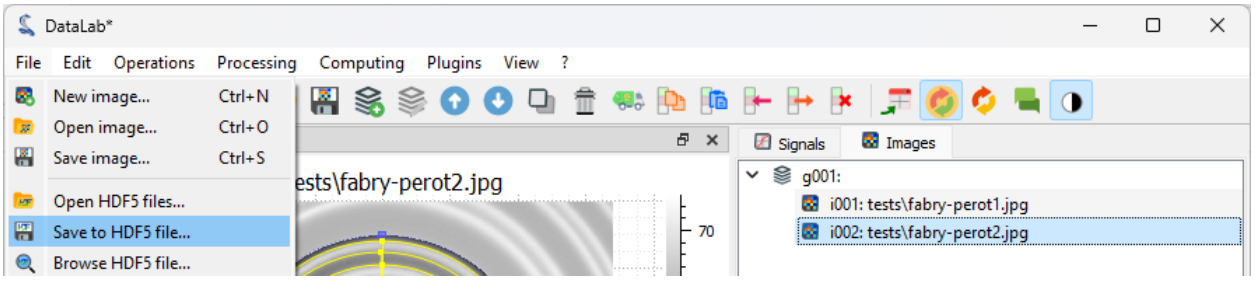


Fig. 66: Save the workspace to a file with "File > Save to HDF5 file…", or the button in the toolbar.

Measuring Laser Beam Size

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

First, we open DataLab and load the images:

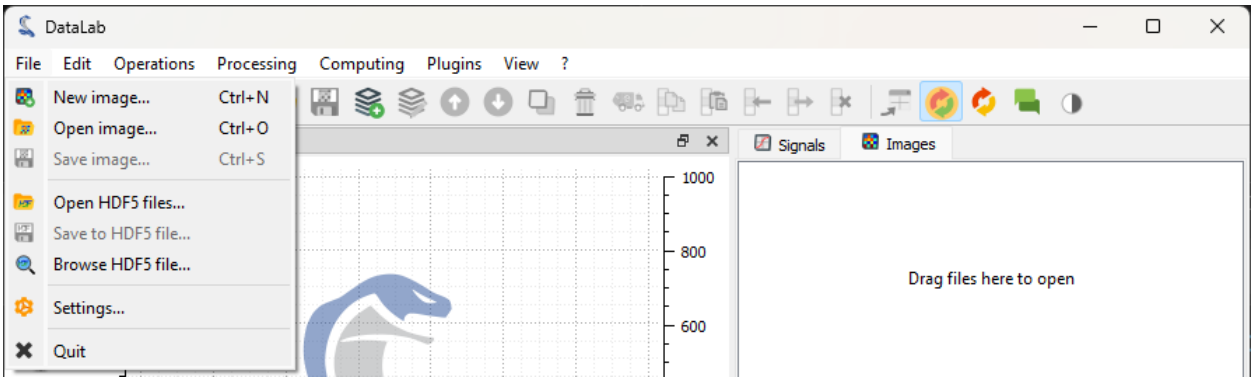


Fig. 67: Open the image files with "File > Open…", or with the button in the toolbar, or by dragging and dropping the files into DataLab (on the panel on the right).

The selected images are loaded in the "Images" panel. The last image is displayed in the main window. On each image, 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.

Note: If we want to display the images side by side, we can select the "Distribute on a grid" entry in the "Operations" menu.

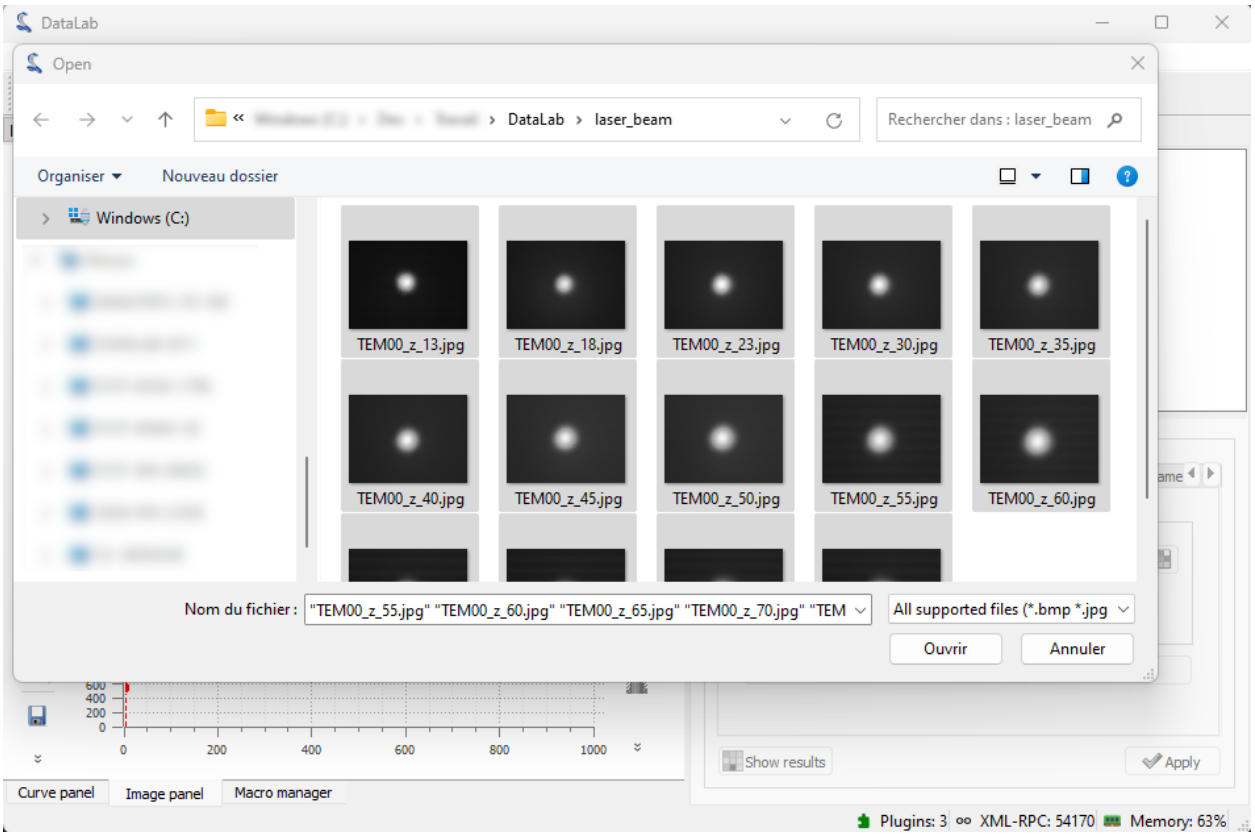


Fig. 68: Select the test images “TEM00_z_*.jpg” and click “Open”.

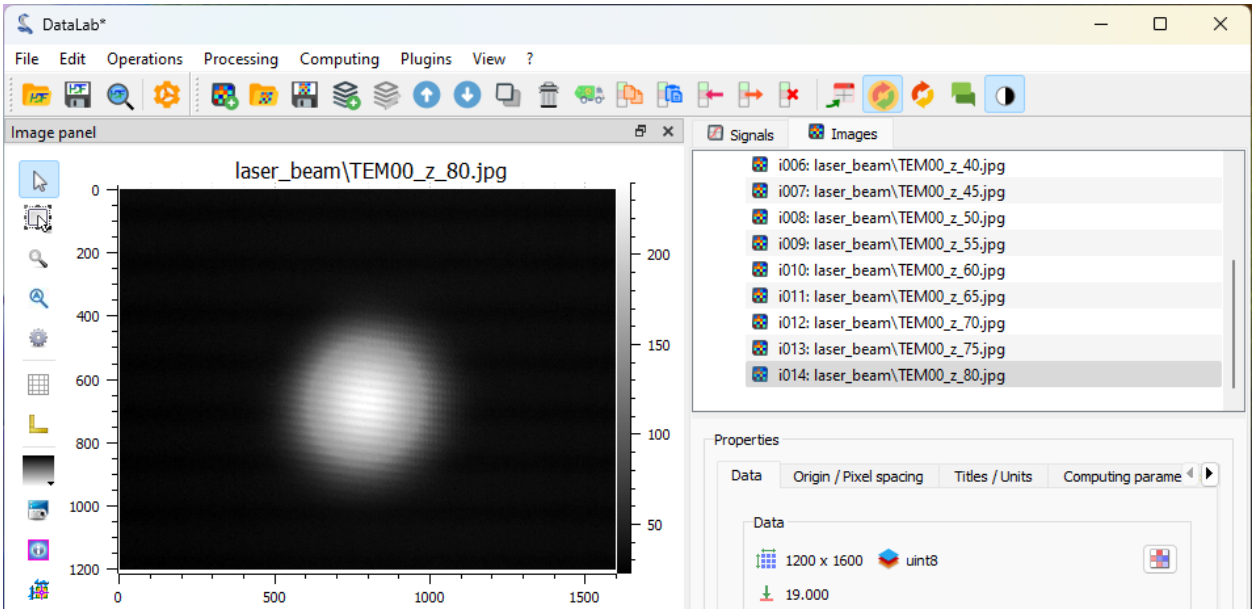


Fig. 69: Zoom in and out with the right mouse button. Pan the image with the middle mouse button.

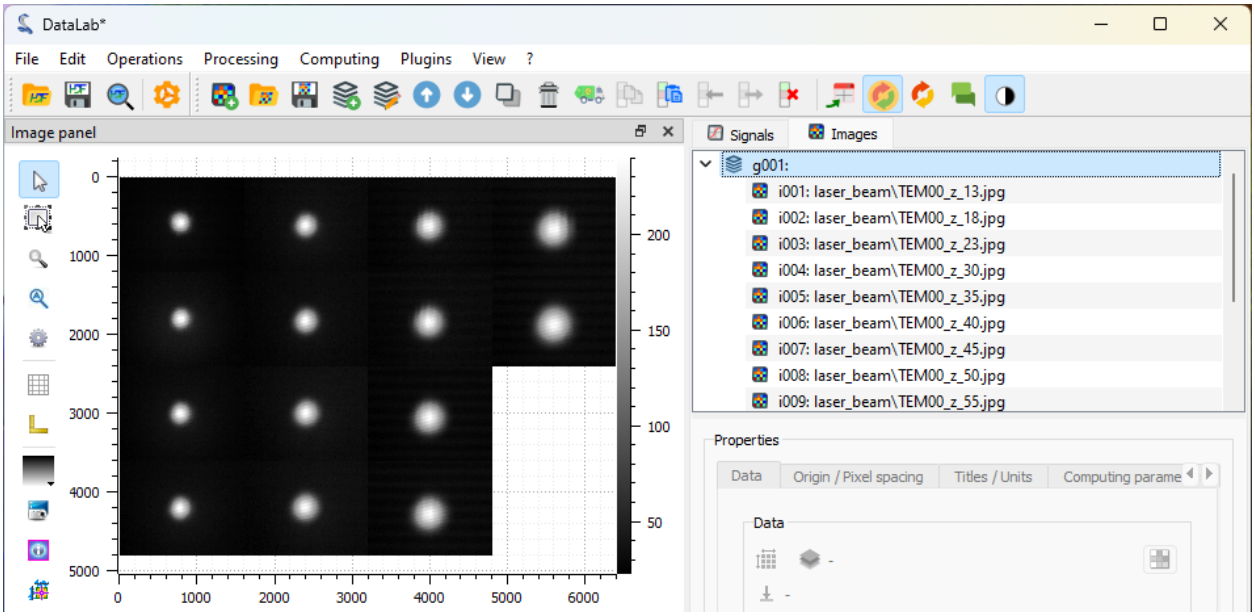


Fig. 70: Images distributed on a 4 rows grid

But, let's go back to the initial display by selecting the "Reset image positions" entry in the "Operations" menu.

If we select on of the images, we can see that there is background noise, so it might be useful to apply a threshold to the images.

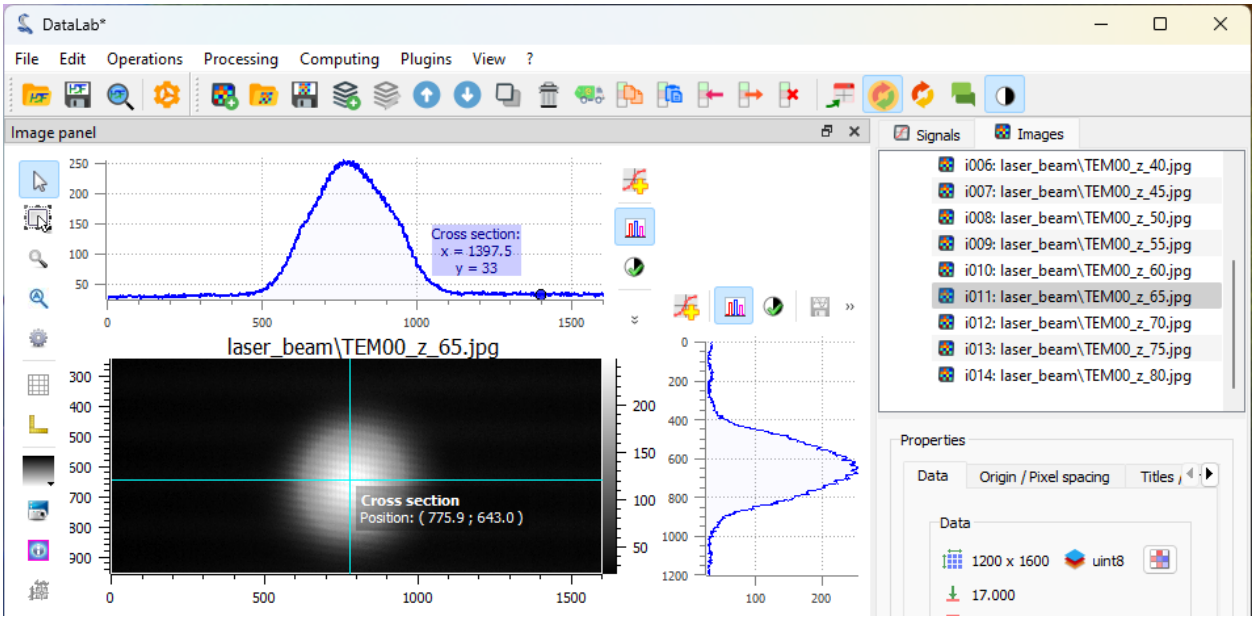


Fig. 71: Select one of the images in the "Images" panel, select the associated image in the visualization panel, and enable the "Cross section" tool in the vertical toolbar on the left of the visualization panel (this tool is a PlotPy feature). On this figure, we can see that the background noise is around 30 lsb (to show the curve marker, we had to select the profile curve and right-click on it to display the context menu, and select "Markers > Bound to active item").

Now, let's try another method to measure the beam size.

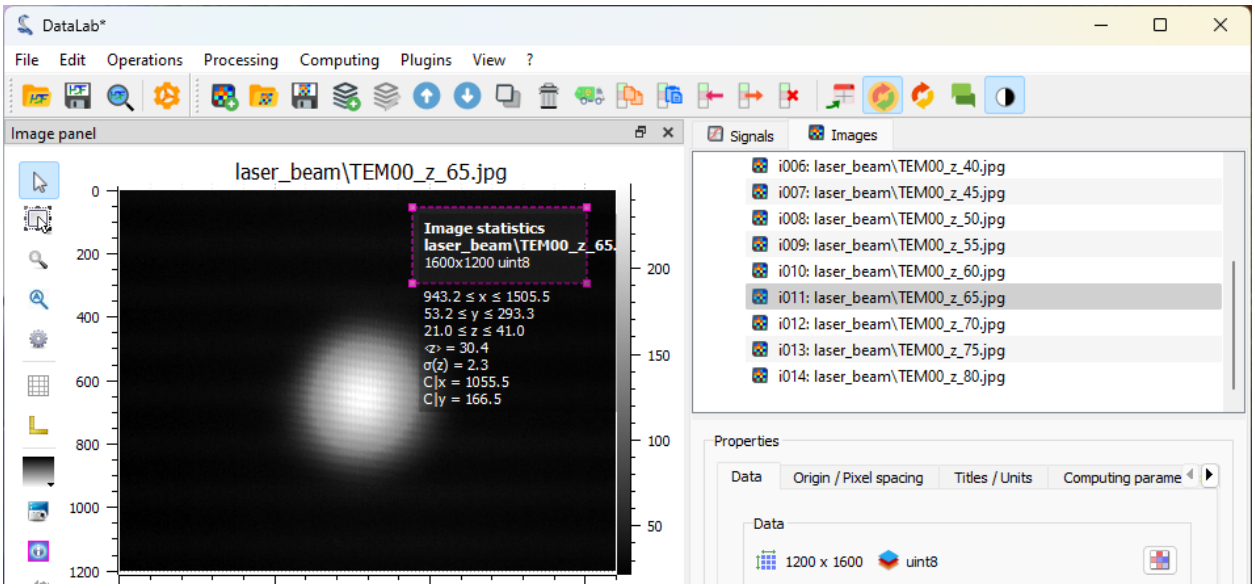


Fig. 72: Another way to measure the background noise is to use the "Image statistics" tool in the vertical toolbar on the left of the visualization panel. It displays statistics on a the rectangular area defined by dragging the mouse on the image. This confirms that the background noise is around 30 lsb.

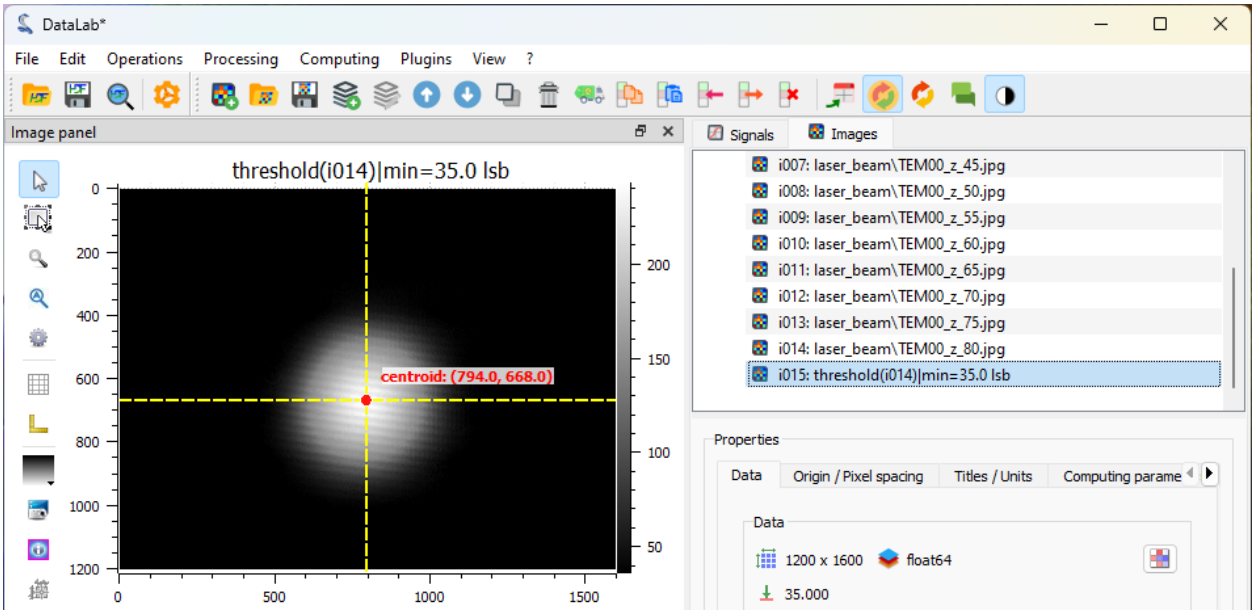


Fig. 73: After applying a threshold at 35 lsb (with "Processing > Thresholding…"), we can compute a more accurate position of the beam center using "Computing > Centroid".

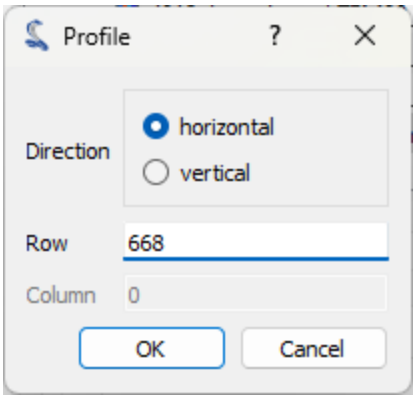


Fig. 74: Then we can extract a line profile along the horizontal axis with "Operations > Intensity profiles > Line profile". We set the row position to the centroid position computed previously (i.e. 668).

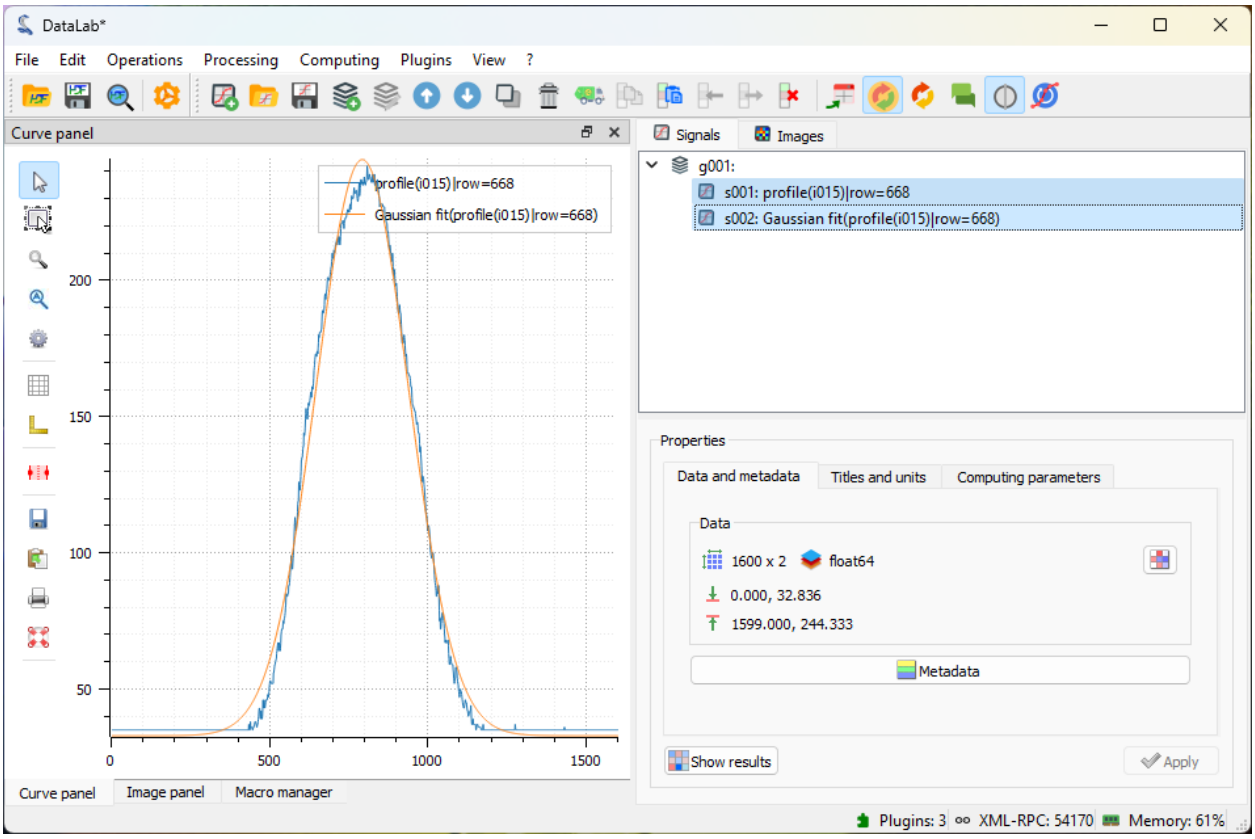


Fig. 75: The intensity profile is displayed in the "Signals" panel. We can fit the profile to a Gaussian function with "Processing > Fitting > Gaussian fit". Here we have selected both signals.

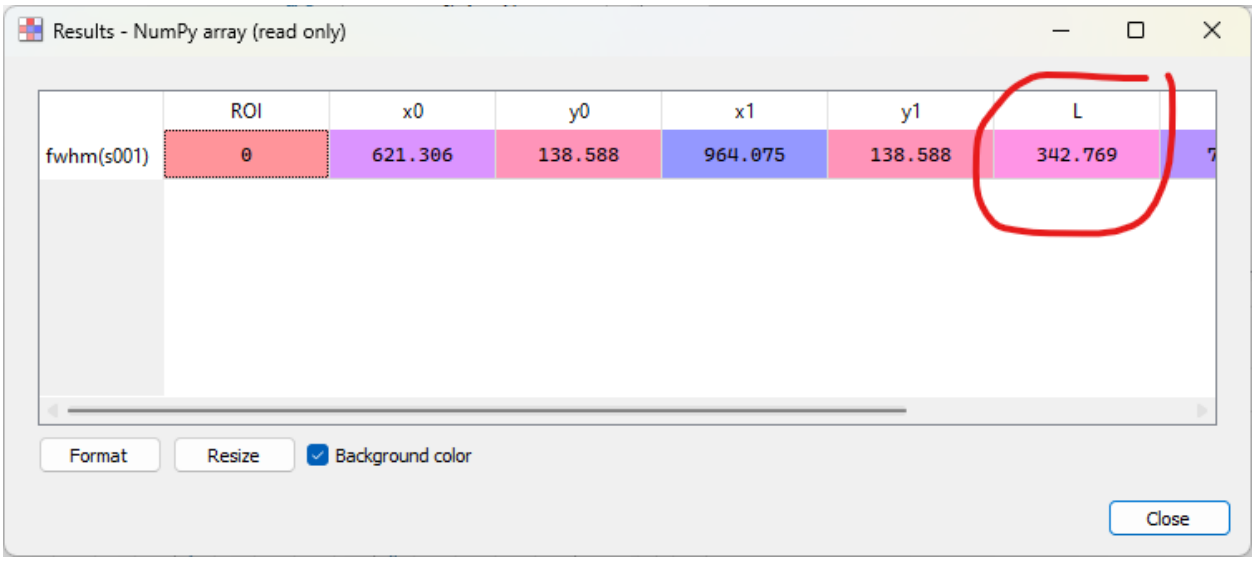


Fig. 76: If we go back to the first signal, the intensity profile, we can also directly compute the FWHM with "Computing > Full width at half maximum". The "Results" dialog displays a lot of information about the computation, including the FWHM value (that is the *L* column, "L" for "Length" because the result shape is a segment and FWHM is the length of the segment).

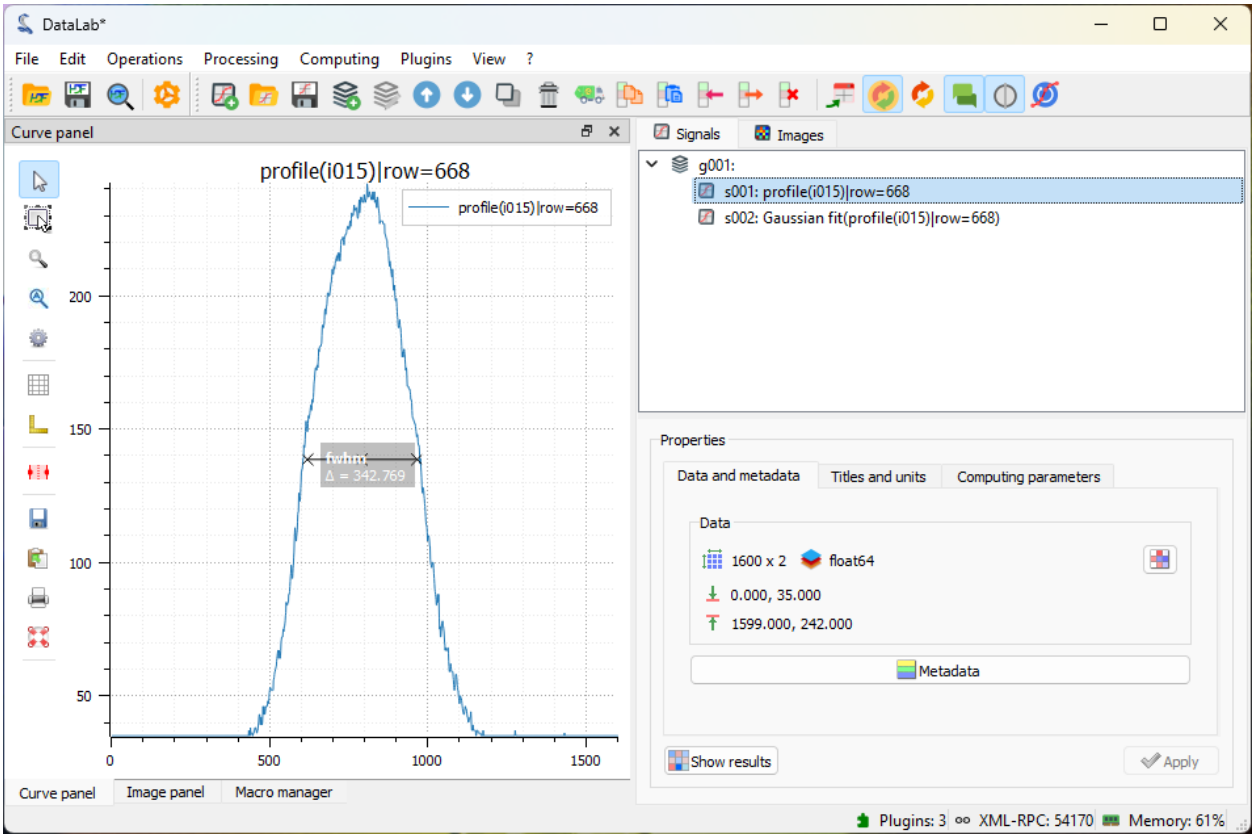


Fig. 77: The FWHM is also displayed on the curve, with an optional label (here, the title of this measurement has been displayed with "View > Show graphical object titles" or the button in the toolbar).

From the "Images" panel, we can extract the radial intensity profile with "Operations > Intensity profiles > Radial profile".

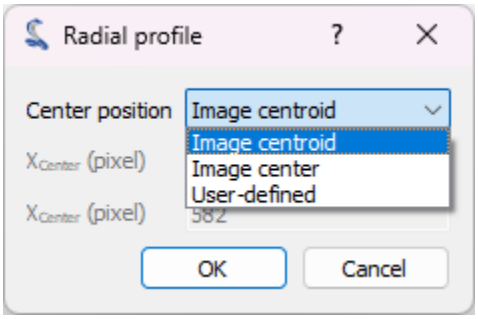


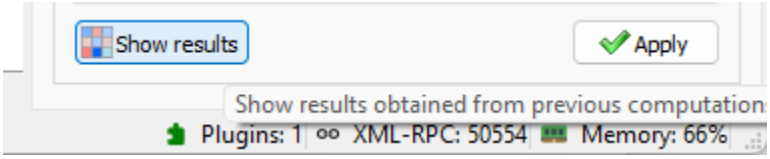
Fig. 78: The radial intensity profile may be computed around the centroid position, or around the center of the image, or around a user-defined position. Here we have selected the centroid position.

All these operations and computations that we have performed on a single image can be applied to all the images in the "Images" panel.

To do that, we begin by cleaning the "Signals" panel (with "Edit > Delete all" or the button in the toolbar). We alse clean the intermediate results in the "Images" panel by selecting the images obtained during our prototyping and deleting them individually (with "Edit > Remove" or the button).

Then, we select all the images in the "Images" panel (individually, or by selecting the whole group "g001").

Note: If you want to show the computing results again, you can select the "Show results" entry in the "Computing" menu, or the "Show results" button, below the image list:



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 computing results, the visualization settings (colormaps, contrast, etc.), the metadata, and the annotations.

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

Prototyping a custom processing pipeline

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

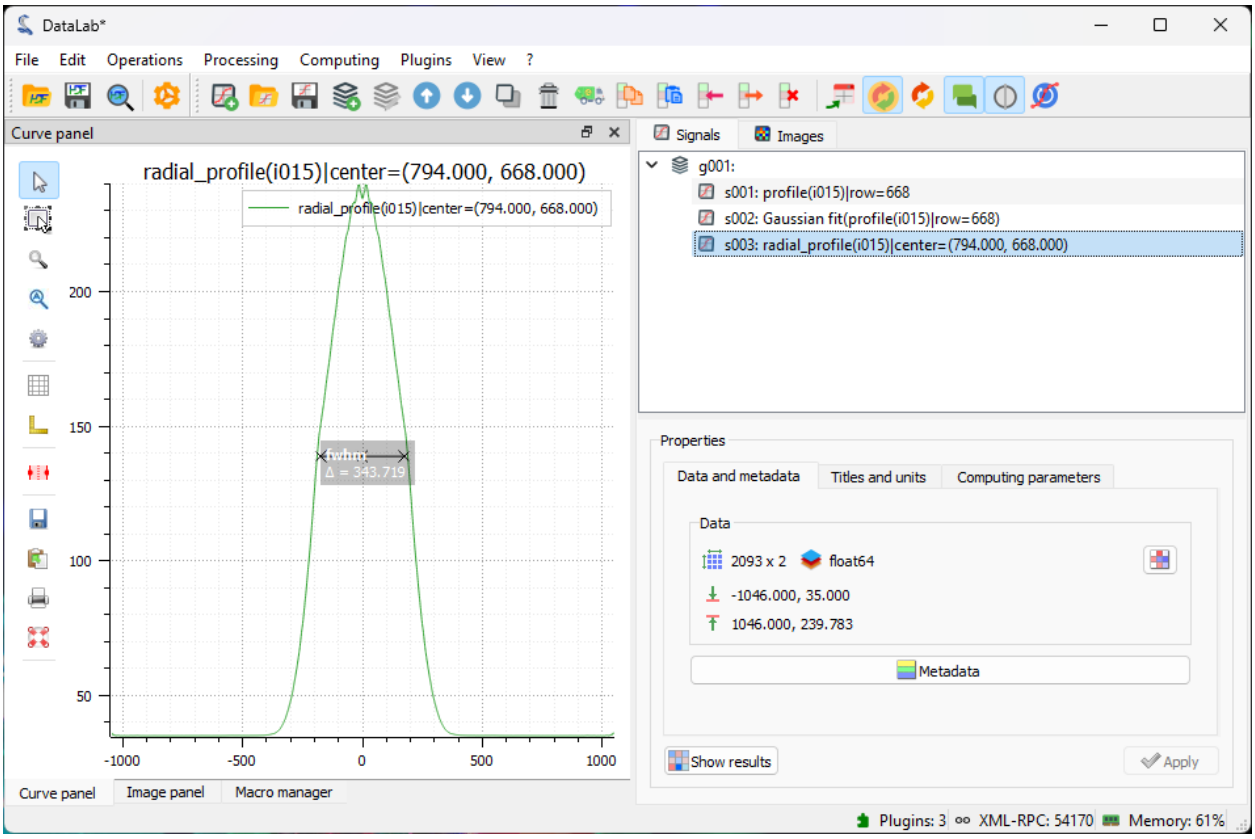


Fig. 79: The radial intensity profile is displayed in the "Signals" panel. It is smoother than the line profile, because it is computed from a larger number of pixels, thus averaging the noise.

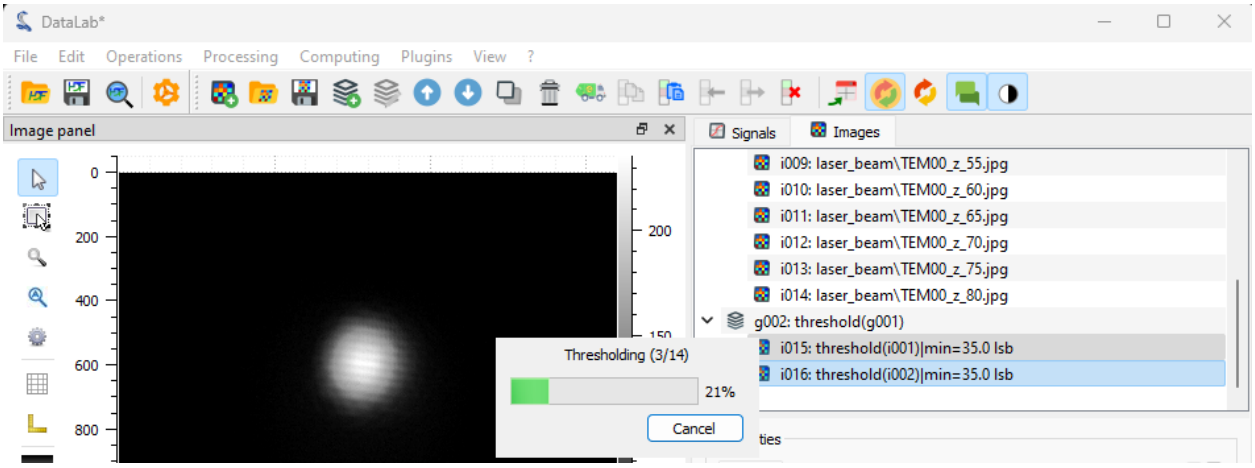


Fig. 80: We apply the threshold to all the images, and the we extract the radial intensity profile for all the images (after selecting the whole group "g002" - it should be automatically selected if you had selected "g001" before applying the threshold).

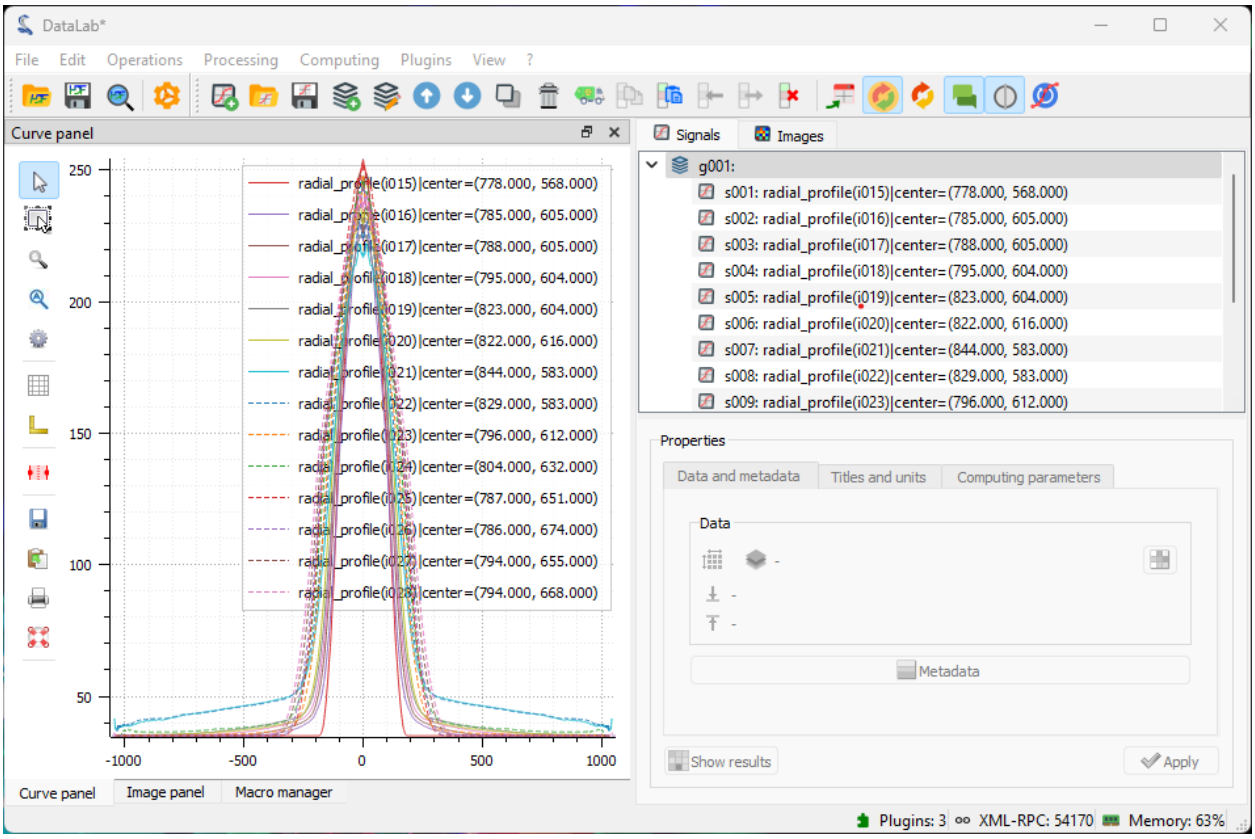


Fig. 81: The "Signals" panel now contains all the radial intensity profiles.

	ROI	x0	y0	x1	y1	L	Xc	Yc
fwhm(s001)	0	-90.146	146.086	90.1454	146.086	180.291	-0.000307247	146.086
fwhm(s002)	0	-97.5273	134.95	97.5273	134.95	195.055	-2.84217e-14	134.95
fwhm(s003)	0	-102.844	144.563	102.845	144.563	205.689	0.000226296	144.563
fwhm(s004)	0	-108.246	140.839	108.246	140.839	216.492	-1.66472e-05	140.839
fwhm(s005)	0	-114.898	134.226	114.898	134.226	229.796	-1.0366e-05	134.226
fwhm(s006)	0	-120.149	138.229	120.147	138.229	240.295	-0.000828078	138.229
fwhm(s007)	0	-132.471	134.219	132.471	134.219	264.942	2.84217e-14	134.219
fwhm(s008)	0	-136.738	138.216	136.737	138.216	273.475	-0.000167016	138.216
fwhm(s009)	0	-136.727	139.425	136.727	139.425	273.454	-0.000201918	139.425

Fig. 82: We can compute the FWHM of all the radial intensity profiles: the "Results" dialog displays the FWHM values for all the profiles.

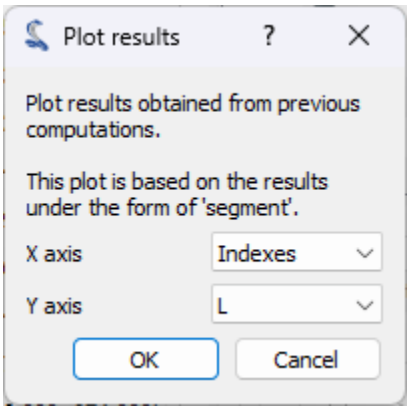


Fig. 83: Finally, we can plot the beam size as a function of the position along the propagation axis. To do that, we use the "Plot results" feature in the "Computing" menu. This feature allows to plot result data sets by choosing the x and y axes among the result columns. Here, we choose the to plot the FWHM values (L) as a function of the image index (*Indexes*).

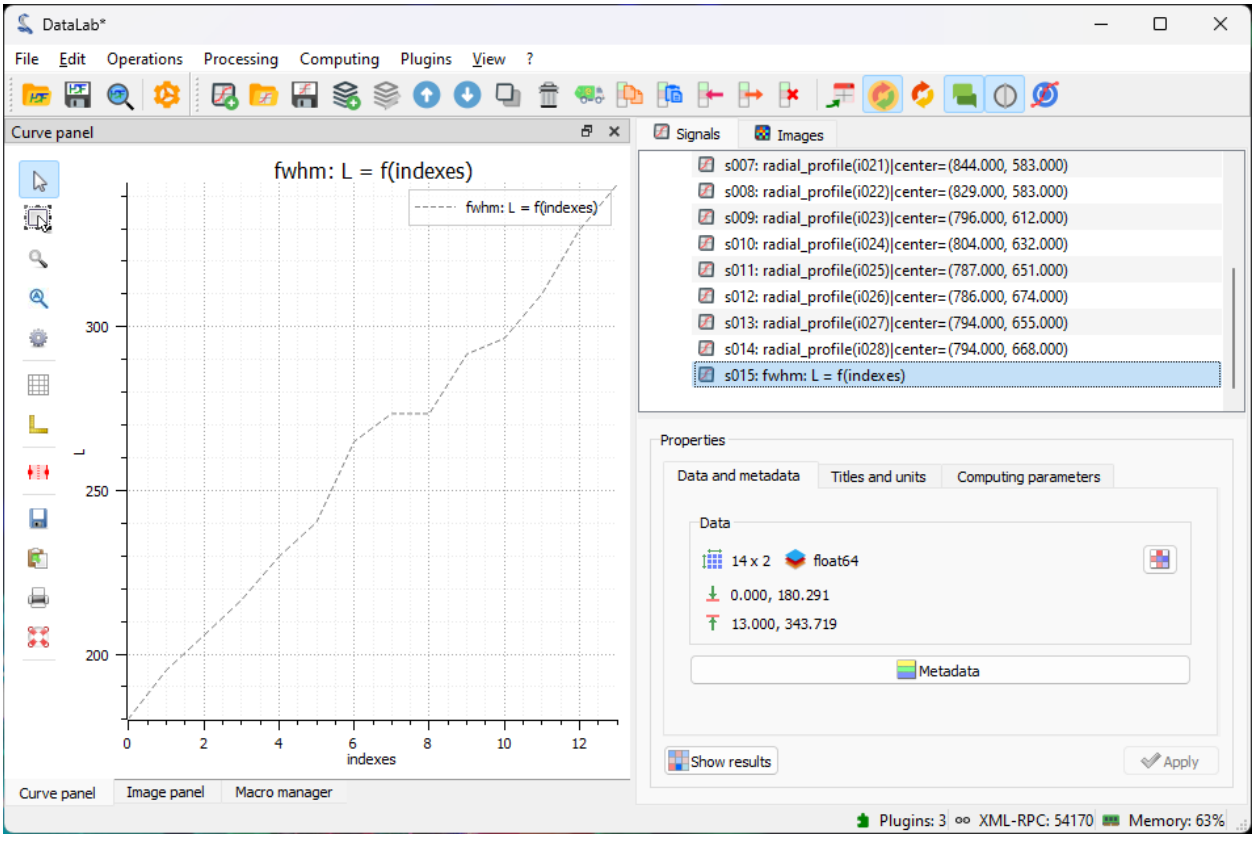


Fig. 84: The plot is displayed in the "Signals" panel and shows that the beam size increases with the position along the propagation axis (the position is here in arbitrary units, the image index).

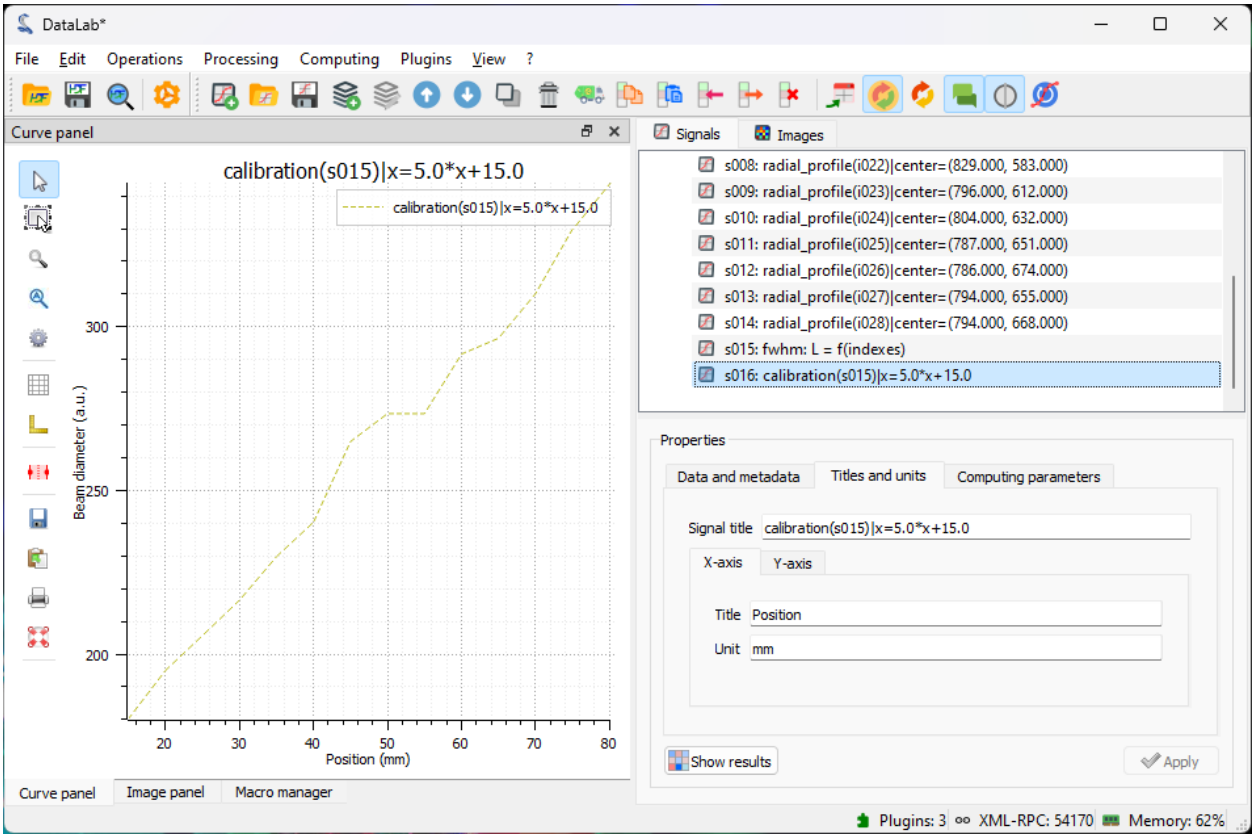


Fig. 85: We can also calibrate the X and Y axis using “Processing > Linear calibration”. Here we have set the X axis to the position in mm (and entered the title and unit in the “Properties” group box).

Define a custom processing function

For illustrating 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.

Here is the code of the `weighted_average_denoise` function:

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

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

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

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

For testing our processing function, we will use a generated image from a DataLab plugin example (*plugins/examples/cdl_example_imageproc.py*). Before starting, make sure that the plugin is installed in DataLab (see the first steps of the tutorial *Detecting blobs on an image*).

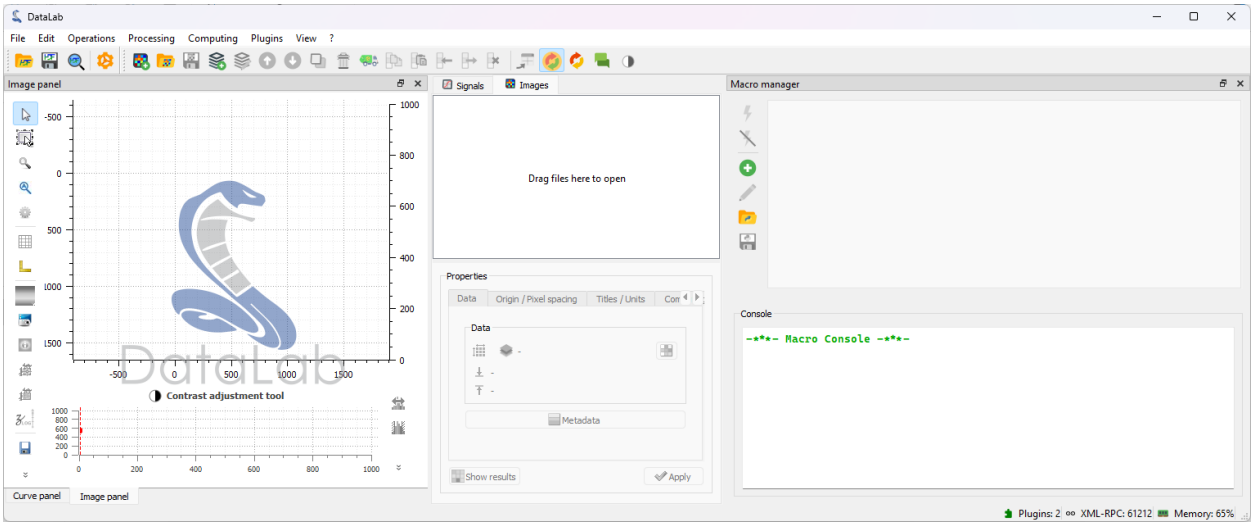


Fig. 86: To begin, we reorganize the window layout of DataLab to have the "Image Panel" on the left and the "Macro Panel" on the right.

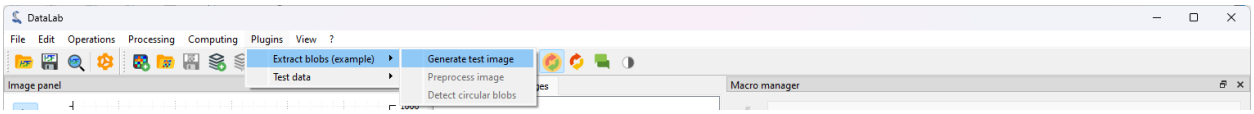


Fig. 87: We generate a new image using the “Plugins > Extract blobs (example) > Generate test image” menu.

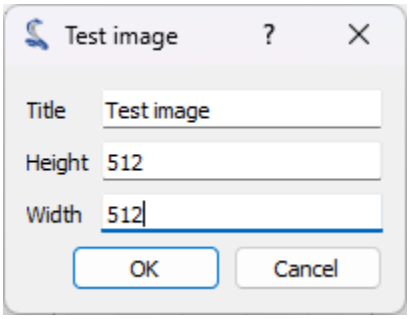


Fig. 88: We select a limited size for the image (e.g. 512x512 pixels) because our algorithm is quite slow, and click on “OK”.

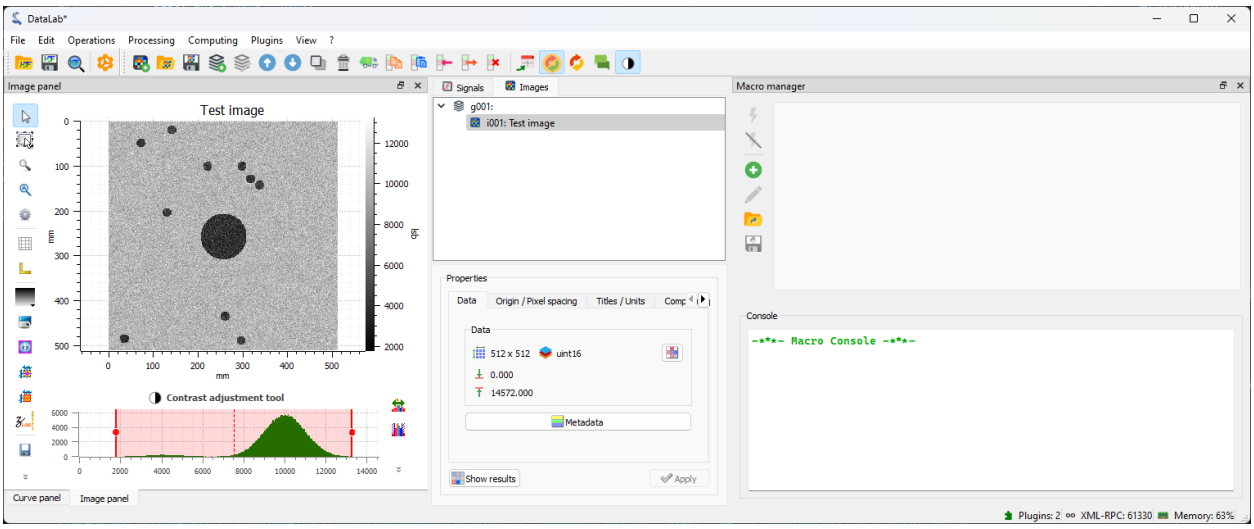


Fig. 89: We can now see the generated image in the “Image Panel”.

Create a macro-command

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". 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 cdl.proxy import RemoteProxy

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

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

# Initialize the proxy to DataLab
proxy = RemoteProxy()

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

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

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

In DataLab, macro-commands are simply Python scripts:

- 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 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.

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.

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

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

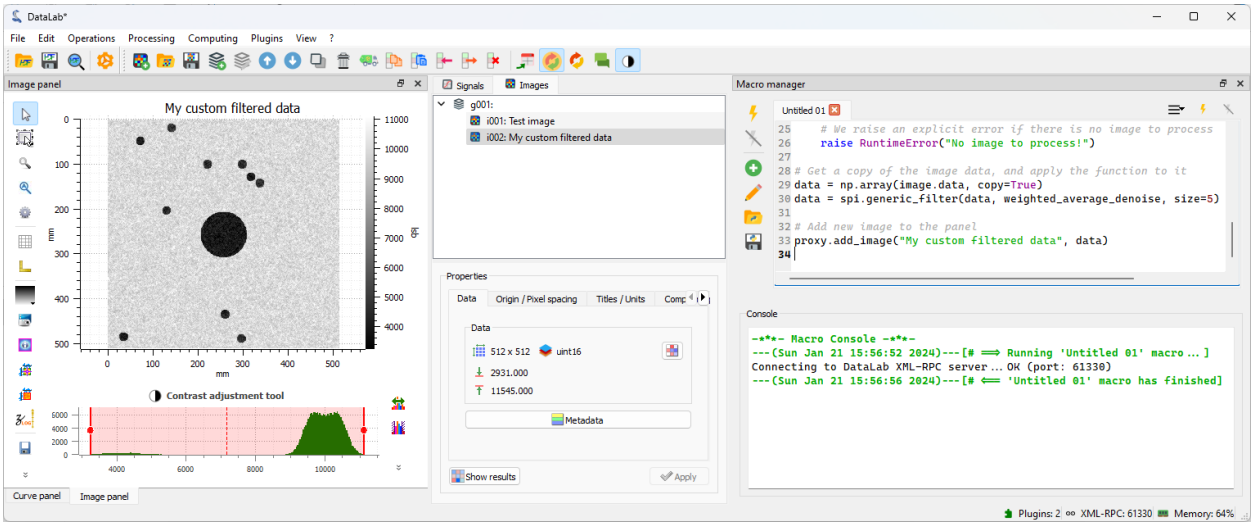


Fig. 90: When the macro-command has finished, we can see the new image in the "Image Panel". Our filter has been applied to the image, and we can see that the noise has been reduced.

Prototyping with an external IDE

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

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

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

- If you use the stand-alone version of DataLab or if you want or need to keep DataLab and Spyder in separate Python environments, you can install the DataLab Simple Client (`cdl-client`) using the `pip` package manager:

  ```
  pip install cdl-client
  ```

  Or you may also install the DataLab Python package (`cdl`) 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 cdl
  ```

- 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 `cdl` package).

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

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

Created on Fri May 12 12:28:56 2023

@author: p.raybaut
"""

# %% Importing necessary modules

import numpy as np
import scipy.ndimage as spi
from cdlclient import SimpleRemoteProxy

# %% Connecting to DataLab current session

proxy = SimpleRemoteProxy()
proxy.connect()

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


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

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

(continues on next page)

(continued from previous page)

```
    """

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

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


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

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

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

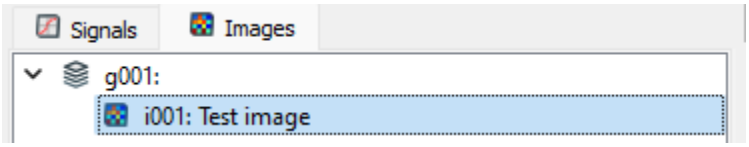


Fig. 91: We go back to DataLab and select the first image in the "Image Panel".

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 only requirement is to install a DataLab client in Jupyter's Python environment (see the previous section for more details: that is exactly the same procedure as for Spyder or any other IDE like Visual Studio Code, for example).

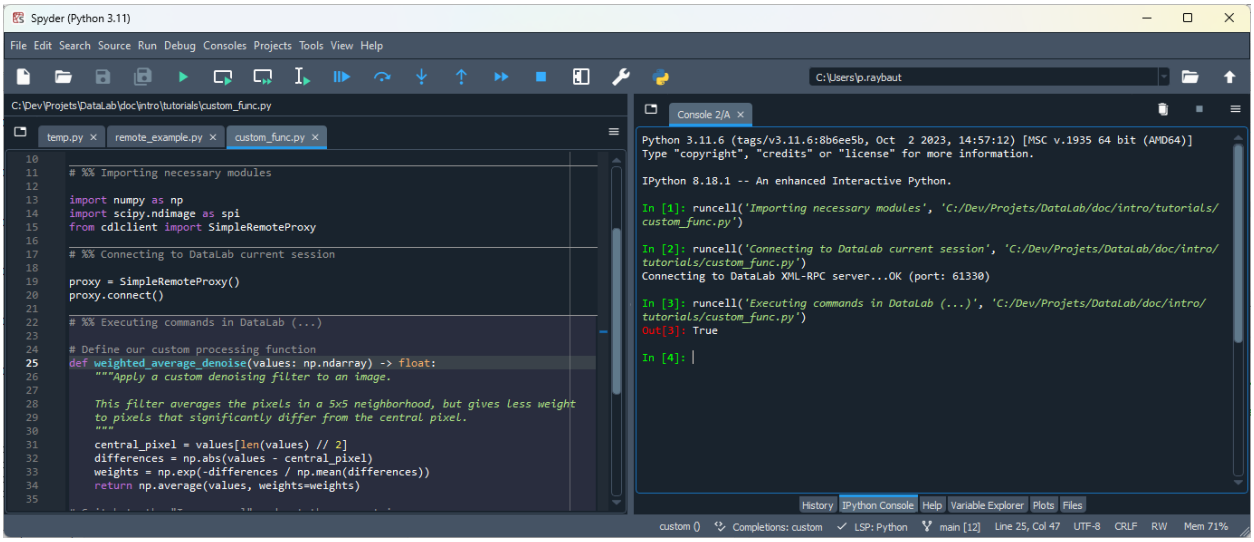


Fig. 92: Then, we execute the script in Spyder, step-by-step (using the defined cells), and we can see the result in DataLab.

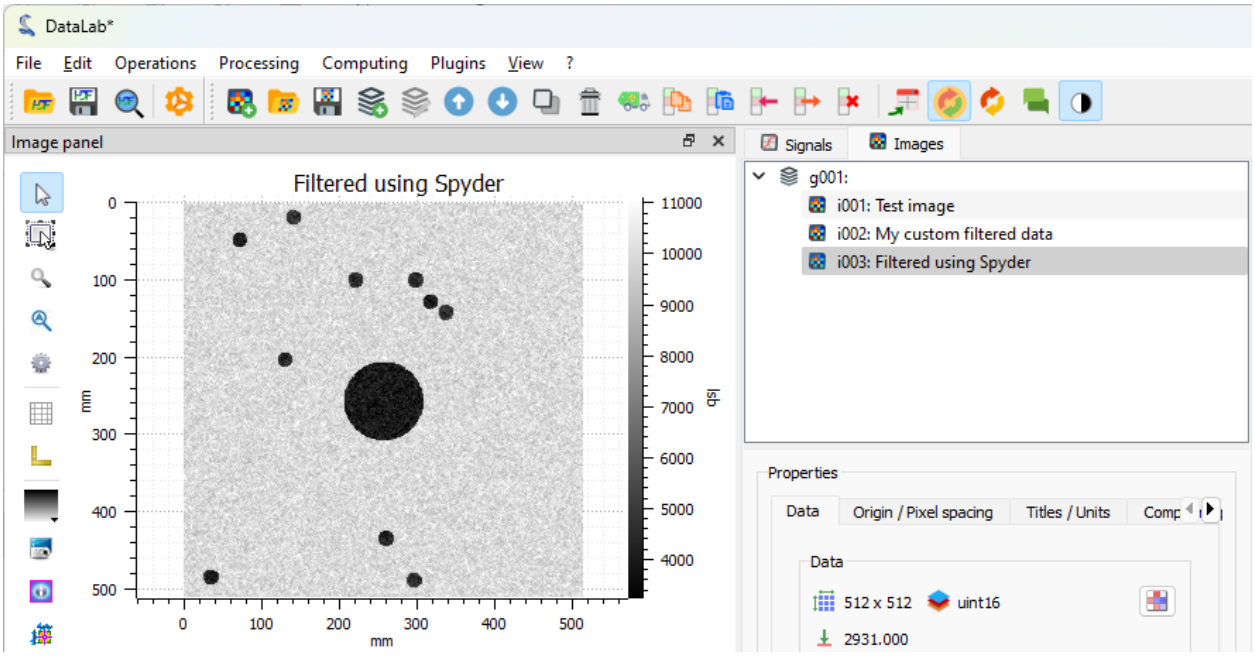


Fig. 93: We can see in DataLab that a new image has been added to the "Image Panel". This image is the result of the execution of the script in Spyder. Here we have used the script without any modification, but we could have modified it to test new ideas, and then use the modified script in DataLab.

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. 94: Once the client is installed, we can start Jupyter and create a new notebook.

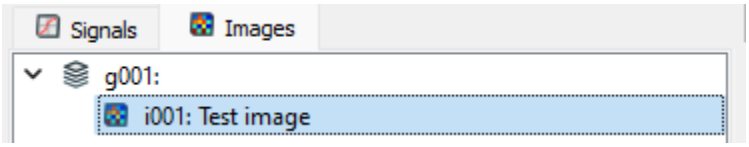


Fig. 95: We go back to DataLab and select the first image in the “Image Panel”.

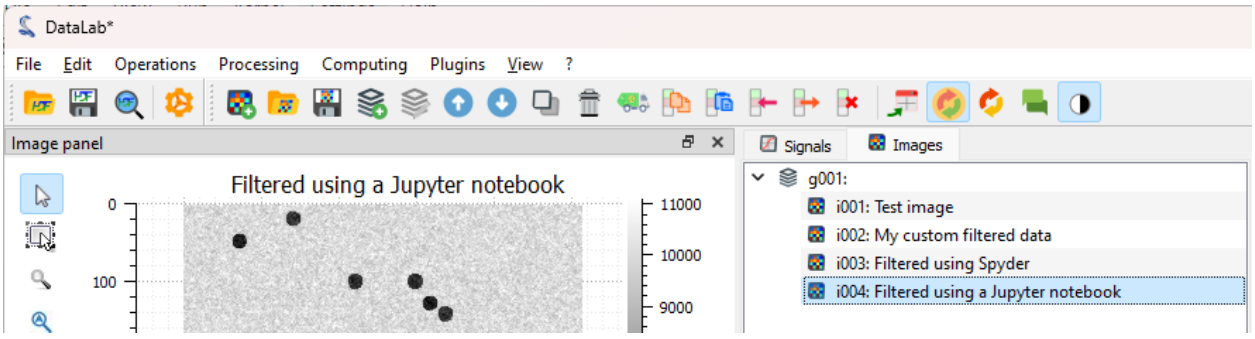


Fig. 96: Then, we execute the notebook in Jupyter, step-by-step (using the defined cells), and we can see the result in DataLab. Once again, we can see in DataLab that a new image has been added to the “Image Panel”. This image is the result of the execution of the notebook in Jupyter. As for the script in Spyder, we could have modified the notebook to test new ideas, and then use the modified notebook in DataLab.

Creating a plugin

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

See also:

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

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

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

The most significant change is that we need to define a function that will be operating on DataLab's native image objects (*cdl.obj.ImageObj*), instead of operating on NumPy arrays. So we need to find a way to call our custom function `weighted_average_denoise` with a *cdl.obj.ImageObj* as input and output. To avoid writing a lot of boilerplate code, we can use the function wrapper provided by DataLab: *cdl.core.computation.image.Wrap11Func*.

Besides we need to define a class that describes our plugin, which must inherit from *cdl.plugins.PluginBase* and name the Python script that contains the plugin code with a name that starts with `cdl_` (e.g. `cdl_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:

```
# -*- 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.github.io/en/features/general/plugins.html>`_).
"""

import numpy as np
import scipy.ndimage as spi

import cdl.core.computation.image as cpi
import cdl.obj
import cdl.param
import cdl.plugins

```

(continues on next page)

(continued from previous page)

```python

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

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

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

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


class CustomFilters(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 instead of NumPy arrays
                wrapped_func = cpi.Wrap11Func(func)
                acth.new_action(
                    name, triggered=lambda: proc.compute_11(wrapped_func, title=name)
                )
```

To test it, we have to add the plugin script to one of the plugin directories that are discovered by DataLab at startup (see the *Plugins* section for more details, or the *Detecting blobs on an image* for an example).

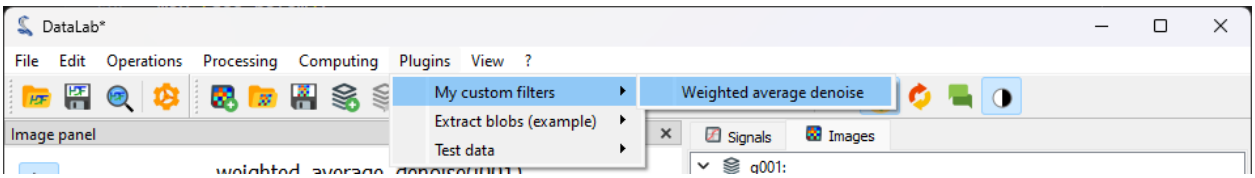


Fig. 97: We restart DataLab and we can see that the plugin has been loaded.

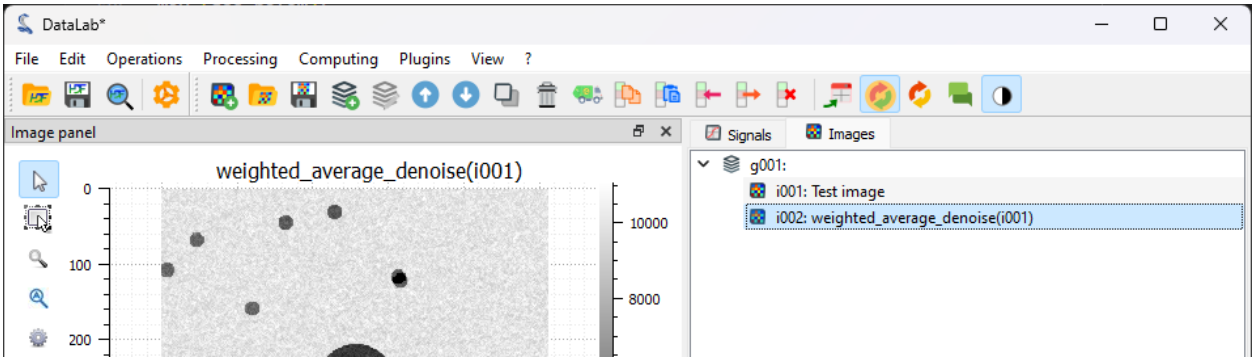
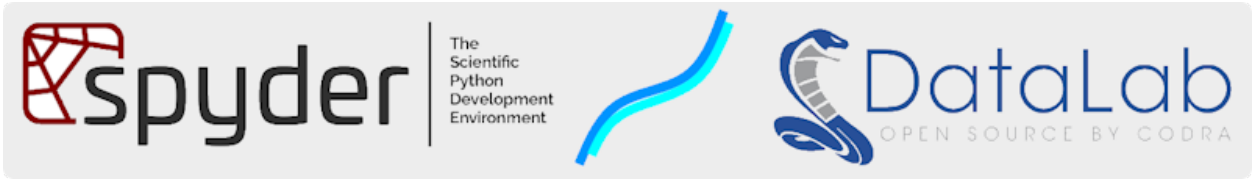


Fig. 98: We generate again our test image using (see the first steps of the tutorial), and we process it using the plugin: "Plugins > My custom filters > Weighted average denoise".

DataLab and Spyder: a perfect match

This tutorial shows how to use Spyder to work with DataLab through an example, using fake algorithms and data that represent an hypothetical research/technical work. The goal is to illustrate how to use DataLab to test your algorithms with some data, and how to debug them if necessary.

The example is quite simple, but it illustrates the basic concepts of working with 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 tool that may be used to 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 a software to process data (signals or images): this software may either be a stand-alone application or a library that you will use in other applications, or even a simple script that you will run from the command line. In any case, you will need to follow a development process that will include 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 some 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 that 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 paragraphs because the DataLab documentation already provides a lot of information about it.

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

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

Besides, we assume that you have already installed Spyder and that you have started it. If you haven't done it 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 DataLab Simple Client (`pip install cdlclient`) 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 you have developed them in Spyder by writing functions that take some data as input and return some processed data as output. Now, you want to test these algorithms with some data.

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

- `test_my_1d_algorithm`: this function returns some 1D data that will allow you to validate your first algorithm which works on 1D data.
- `test_my_2d_algorithm`: this function returns some 2D data that will allow you to validate your second algorithm which works on 2D data.

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

- First, you need to 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 done through a communication protocol.

Here is how to do it:

```
# %% Connecting to DataLab current session

from cdlclient import SimpleRemoteProxy

proxy = SimpleRemoteProxy()
proxy.connect()

# %% Visualizing 1D data from my work

from my_work import test_my_1d_algorithm

```

(continues on next page)

(continued from previous page)

```
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.py')
Connecting to DataLab XML-RPC server...OK (port: 54577)

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

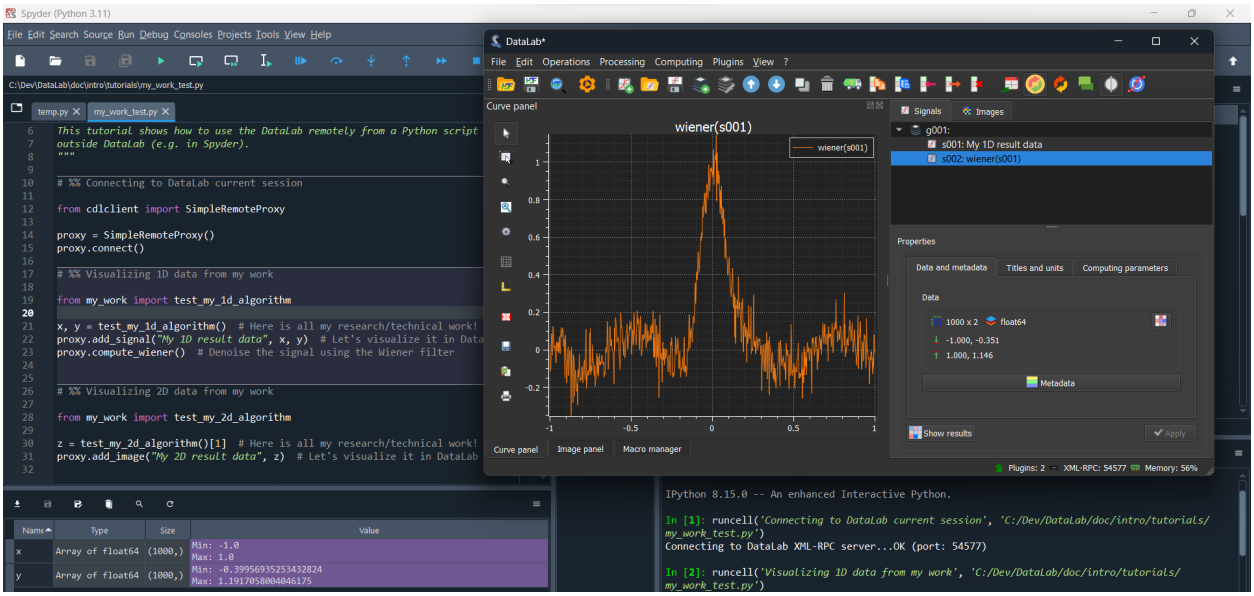


Fig. 99: On this screenshot, we can see the result of evaluating the first two cells: the first cell connects to DataLab, and the second cell visualizes the 1D data returned by the `test_my_1d_algorithm` function.

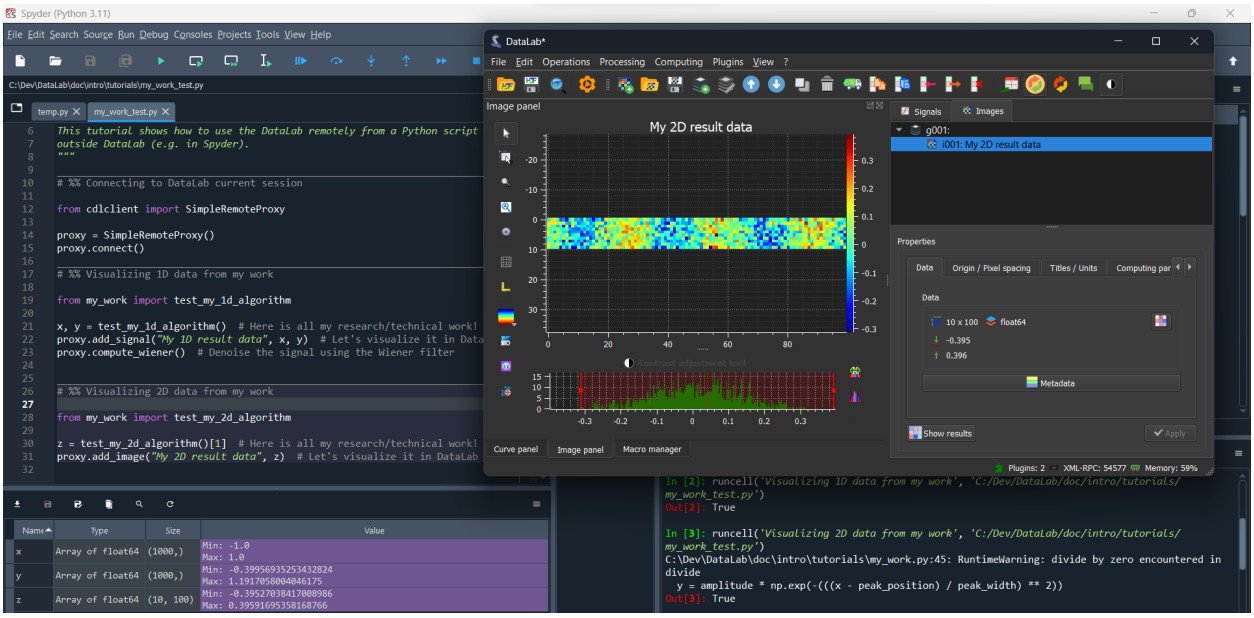


Fig. 100: On this screenshot, we can see the result of evaluating the third cell: the `test_my_2d_algorithm` function returns a 2D array, and we can visualize it directly in DataLab.

Debugging your algorithm with DataLab

Now that you have tested your algorithms with some 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 fake 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 to visualize the data step-by-step 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).
```

(continues on next page)

(continued from previous page)

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

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

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

```
# %% Debugging my work with DataLab

from my_work import generate_2d_data

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

In this simple example, the algorithm is just iterating 10 times and generating a 1D array at each iteration. Each 1D array is then stacked in 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 check that the algorithm is working as expected.

Note: If we had executed the script using 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 is executed in a separate process, we would have been able to manipulate the data in DataLab while the algorithm is paused in Spyder.

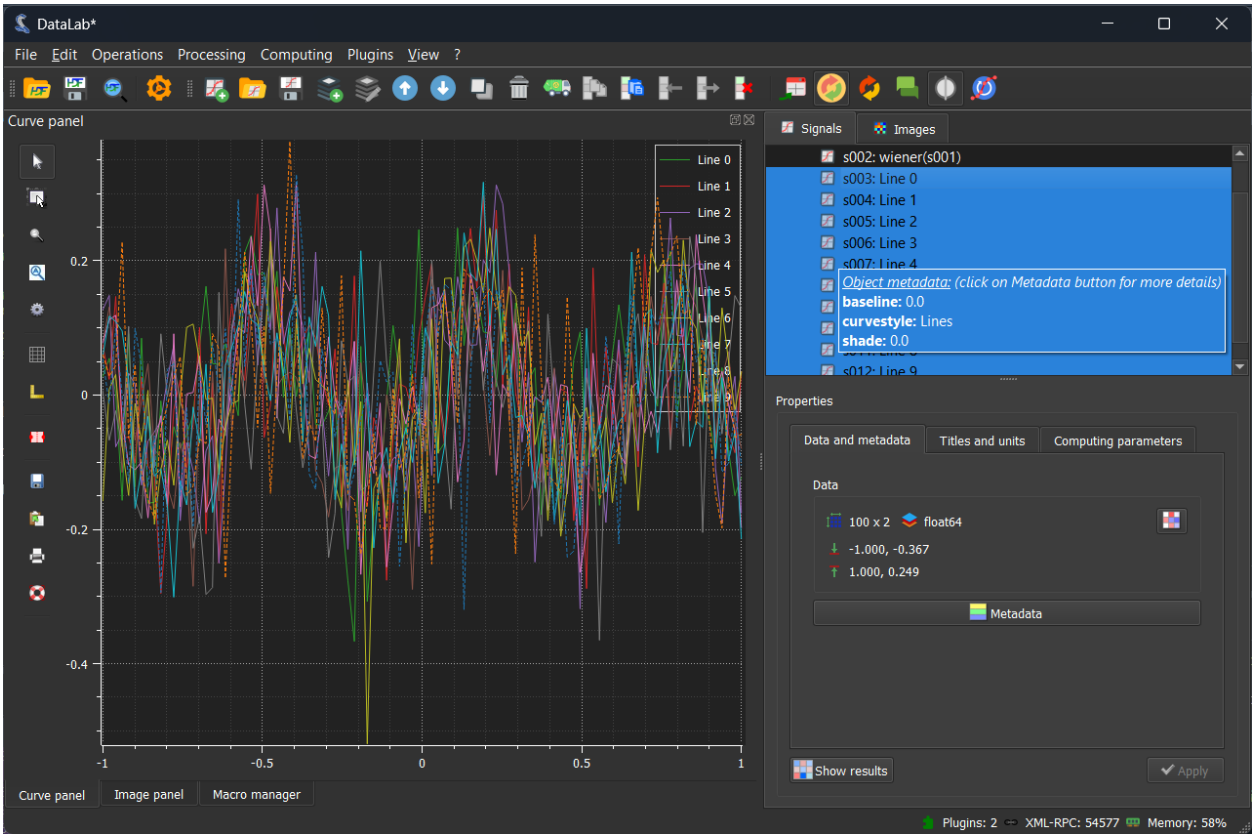


Fig. 101: On this screenshot, we can see the result of evaluating the first cell: the `test_my_2d_algorithm` function is called with the `debug_with_datalab` parameter set to `True`: 10 1D arrays are generated and visualized in DataLab.

CHAPTER

TWO

FEATURES

DataLab features are detailed in the following sections. The first section describes the general features of DataLab, then the two following sections describe the features specific to the signal and image processing panels.

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

See also:

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

2.1 General features

This section describes the general features of DataLab, which concern both the signal and image processing panels.

2.1.1 Workspace

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

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

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

The DataLab **Workspace** is defined as the collection of all data sets which are currently loaded in DataLab, in both the **Signals** and **Images** panels. The workspace may be saved to an HDF5 file, and reloaded later. It is also possible to import data sets from an HDF5 file into the workspace, using the *HDF5 Browser*.

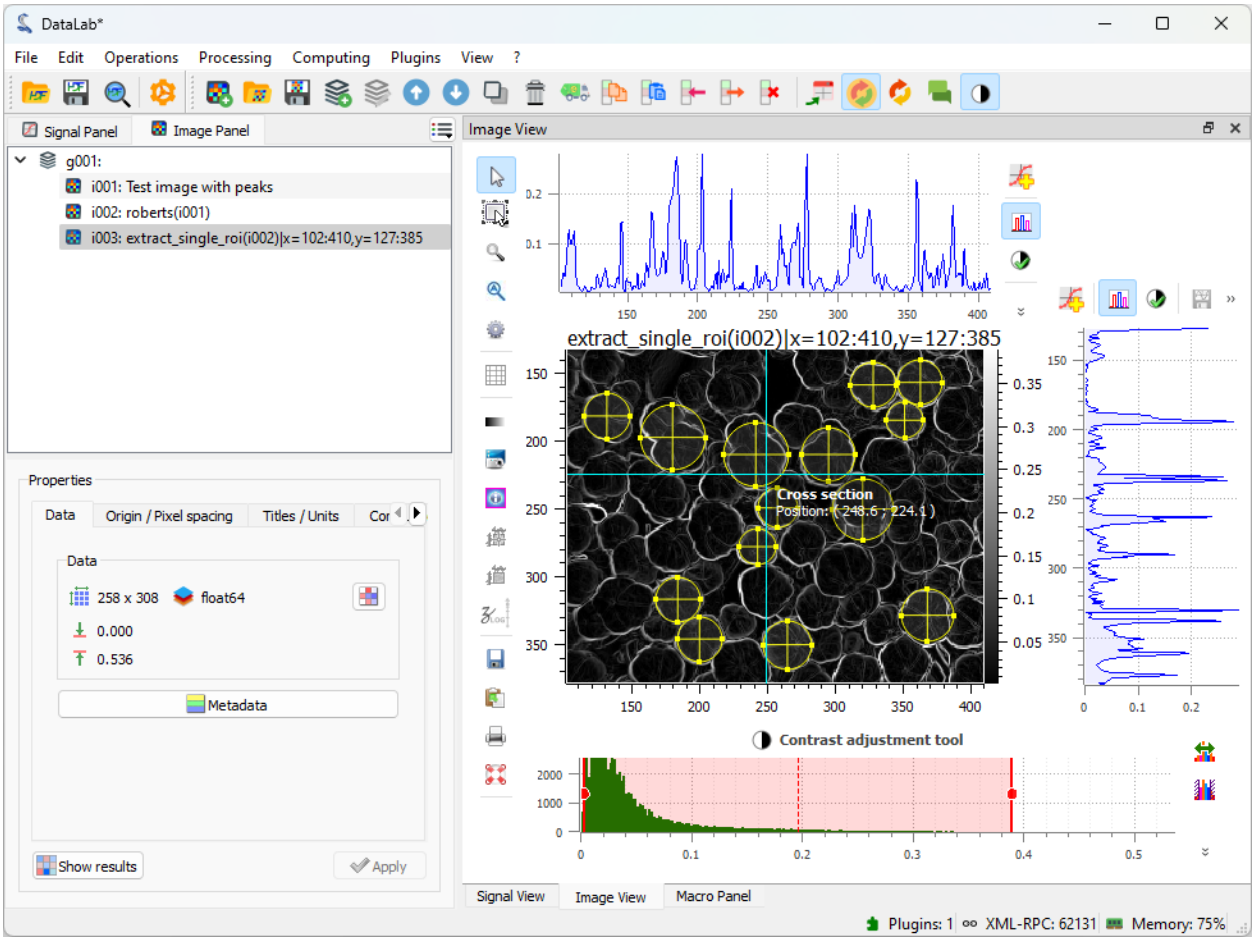


Fig. 1: DataLab main window

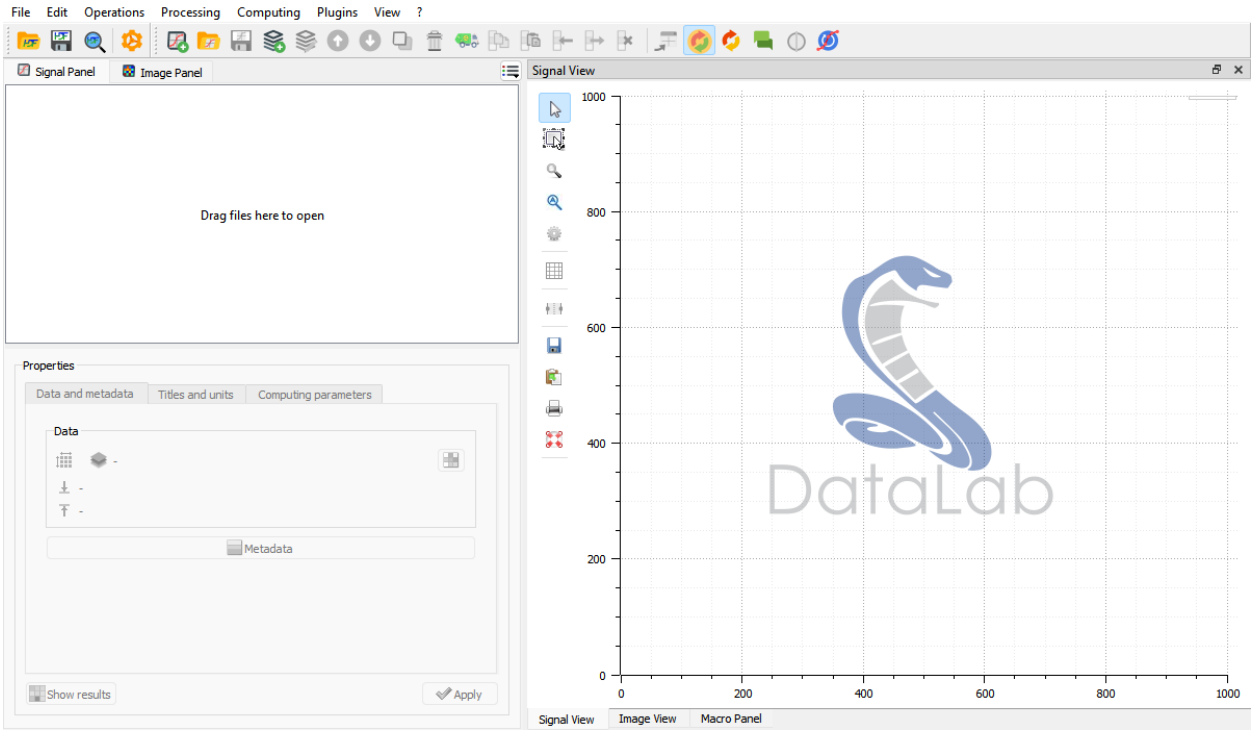
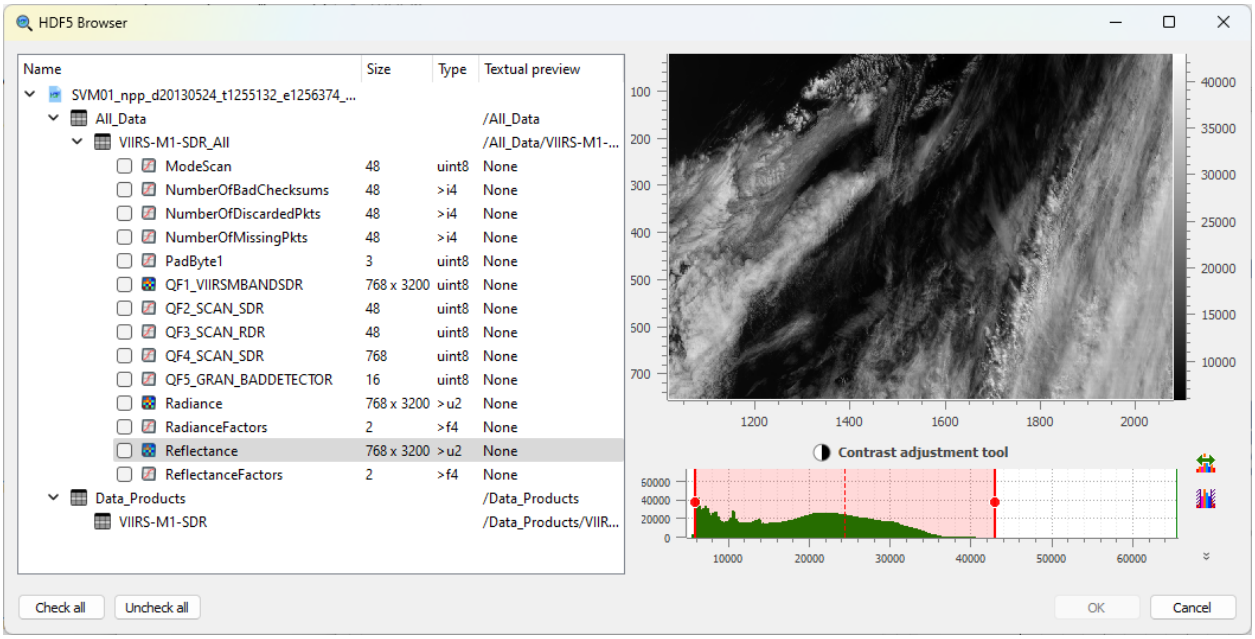


Fig. 2: DataLab main window, at startup.

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

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 HDF5 browser is fairly simple to use:

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

2.1.2 Macros

Overview

There are many ways to extend DataLab with new functionality (see *Plugins* or *Remote controlling*). The easiest way to do so is by using macros. Macros are small Python scripts that can be executed from the “Macro Panel” in DataLab.

Macros can be used to automate repetitive tasks, or to create new functionality. As the plugin and remote control system, macros rely on the DataLab high-level API to interact with the application. This means that you can reuse the same code snippets in macros, plugins, and remote control scripts.

> **Warning:** DataLab handles macros as Python scripts. This means that you can use the full power of Python to create your macros. Even though this is a powerful feature, it also means that you should be careful when running macros from unknown sources, as they can potentially harm your system.

See also:

The DataLab high-level API is documented in the *API* section. The plugin system is documented in the *Plugins* section, and the remote control system is documented in the *Remote controlling* section.

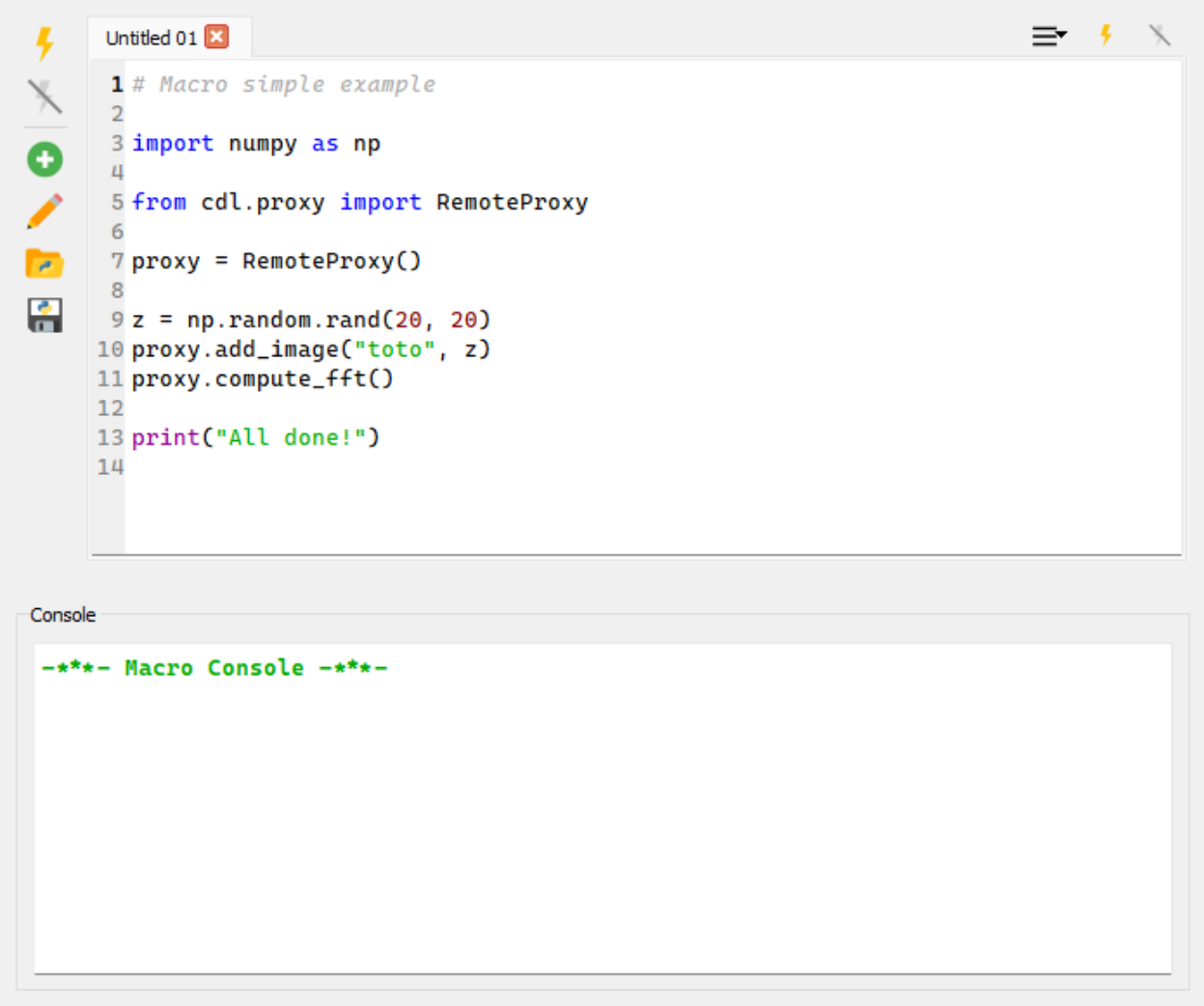


Fig. 3: The Macro Panel in DataLab.

Main features

The Macro Panel is a simple interface to:

- Create new macros, using the "New macro" button.
- Rename existing macros, using the "Rename macro" button.
- Import/export macros from/to files, using the "Import macro" and "Export macro" buttons.
- Execute macros, using the "Run macro" button.
- Stop the execution of a macro, using the "Stop macro" button.

Macros are embedded in the DataLab workspace, so they are saved together with the rest of the data (i.e. with signals and images) when exporting the workspace to a HDF5 file. This means that you can share your macros with other users simply by sharing the workspace file.

Note: Macro are executed in a separate process, so they won't block the main DataLab application. This means that you can continue working with DataLab while a macro is running and that *you can stop a macro at any time* using the button.

Example

For a detailed example of how to create a macro, see the *Prototyping a custom processing pipeline* tutorial.

2.1.3 Remote controlling

DataLab may be controlled remotely using the XML-RPC protocol which is natively supported by Python (and many other languages). Remote controlling allows to access DataLab main features from a separate process.

Note: If you are looking for a lighweight alternative solution to remote control DataLab (i.e. without having to install the whole DataLab package and its dependencies on your environment), please have a look at the DataLab Simple Client package (*pip install cdlclient*).

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 `cdl.proxy`.

From a third-party application

DataLab may also be controlled remotely from a third-party application, for the same purpose.

If the third-party application is written in Python 3, it may directly use the *RemoteProxy* class as mentioned above. From another language, it is also achievable, but it requires to implement a XML-RPC client in this language using the same methods of proxy server as in the *RemoteProxy* class.

Data (signals and images) may also be exchanged between DataLab and the remote client application, in both directions.

The remote client application may be written in any language that supports XML-RPC. For example, it is possible to write a remote client application in Python, Java, C++, C#, etc. The remote client application may be a graphical application or a command line application.

The remote client application may be run on the same computer as DataLab or on a different computer. In the latter case, the remote client application must know the IP address of the computer running DataLab.

The remote client application may be run before or after DataLab. In the latter case, the remote client application must try to connect to DataLab until it succeeds.

Supported features

Supported features are the following:

- Switch to signal or image panel
- Remove all signals and images
- Save current session to a HDF5 file
- Open HDF5 files into current session
- Browse HDF5 file
- Open a signal or an image from file
- Add a signal
- Add an image
- Get object list
- Run calculation with parameters

Note: The signal and image objects are described on this section: *Internal data model*.

Some examples are provided to help implementing such a communication between your application and DataLab:

- See module: `cdl.tests.remoteclient_app`
- See module: `cdl.tests.remoteclient_unit`

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):

```
# -*- 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
"""
```

(continues on next page)

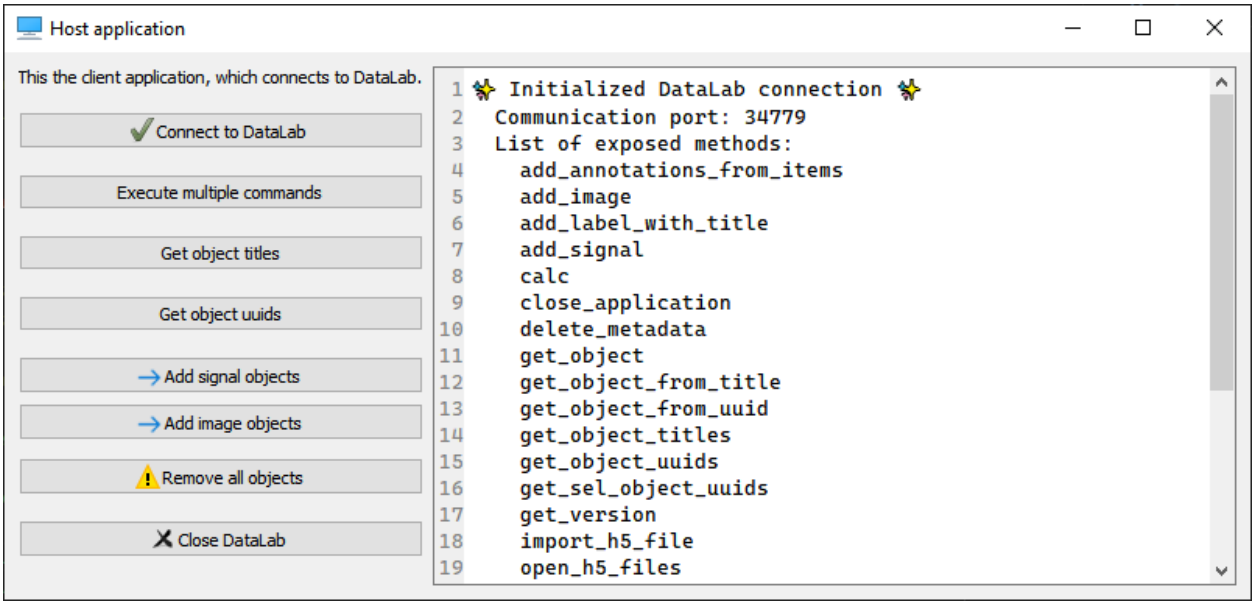


Fig. 4: Screenshot of remote client application test (`cdl.tests.remoteclient_app`)

(continued from previous page)

```
# %% Importing necessary modules

# NumPy for numerical array computations:
import numpy as np

# DataLab remote control client:
from cdl.proxy import 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.compute_derivative()
proxy.toggle_auto_refresh(True)  # Turning on auto-refresh
```

(continues on next page)

(continued from previous page)

```
# %% 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():
    for _idx in range(100):
        proxy.compute_fft()
```

Here is a Python 2.7 reimplementation of this class:

```
# -*- coding: utf-8 -*-
#
# Licensed under the terms of the BSD 3-Clause
# (see cdl/LICENSE for details)

"""
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_cdl_xmlrpc_port():
    """Return DataLab current XML-RPC port"""
    if sys.platform == "win32" and "HOME" in os.environ:
        os.environ.pop("HOME")  # Avoid getting old WinPython settings dir
    fname = osp.join(get_config_dir(), ".DataLab", "DataLab.ini")
    ini = cp.ConfigParser()
    ini.read(fname)
    try:
        return ini.get("main", "rpc_server_port")
    except (cp.NoSectionError, cp.NoOptionError):
```

(continues on next page)

(continued from previous page)

```
        raise ConnectionRefusedError("DataLab has not yet been executed")


class RemoteClient(object):
    """Object representing a proxy/client to DataLab XML-RPC server"""

    def __init__(self):
        self.port = None
        self.serverproxy = None

    def connect(self, port=None):
        """Connect to DataLab XML-RPC server"""
        if port is None:
            port = get_cdl_xmlrpc_port()
        self.port = port
        url = "http://127.0.0.1:" + port
        self.serverproxy = ServerProxy(url, allow_none=True)
        try:
            self.get_version()
        except socket.error:
            raise ConnectionRefusedError("DataLab is currently not running")

    def get_version(self):
        """Return DataLab 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"""
```

(continues on next page)

(continued from previous page)

```
        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"""
        self.serverproxy.import_h5_file(filename, reset_all)

    def open_object(self, filename):
        """Open object from file in current panel (signal/image)"""
        self.serverproxy.open_object(filename)

    def add_signal(
        self, title, xdata, ydata, xunit=None, yunit=None, xlabel=None, ylabel=None
    ):
        """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)

    def add_image(
        self,
        title,
        data,
        xunit=None,
        yunit=None,
        zunit=None,
        xlabel=None,
        ylabel=None,
        zlabel=None,
    ):
        """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)

    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):
```

(continues on next page)

(continued from previous page)

```
        """Get object (signal/image) list for current panel"""
        return self.serverproxy.get_object_uuids(panel)


def test_remote_client():
    """DataLab Remote Client test"""
    cdl = RemoteClient()
    cdl.connect()
    data = np.array([[3, 4, 5], [7, 8, 0]], dtype=np.uint16)
    cdl.add_image("toto", data)


if __name__ == "__main__":
    test_remote_client()
```

Connection dialog

The DataLab package also provides a connection dialog that may be used to connect to a running DataLab instance. It is exposed by the `cdl.widgets.connection.ConnectionDialog` class.

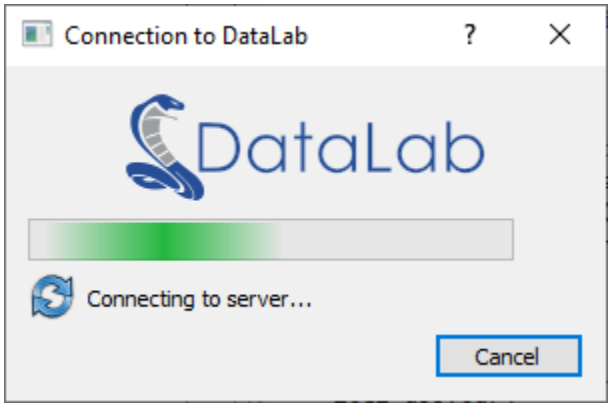


Fig. 5: Screenshot of connection dialog (`cdl.widgets.connection.ConnectionDialog`)

Example of use:

```
# -*- coding: utf-8 -*-
#
# Licensed under the terms of the BSD 3-Clause
# (see cdlclient/LICENSE for details)

"""
DataLab Remote client connection dialog example
"""

# guitest: show,skip

from guidata.qthelpers import qt_app_context
from qtpy import QtWidgets as QW
```

(continues on next page)

(continued from previous page)

```
from cdl.proxy import RemoteProxy
from cdl.widgets.connection import ConnectionDialog


def test_dialog():
    """Test connection dialog"""
    proxy = RemoteProxy(autoconnect=False)
    with qt_app_context():
        dlg = ConnectionDialog(proxy.connect)
        if dlg.exec():
            QW.QMessageBox.information(None, "Connection", "Successfully connected")
        else:
            QW.QMessageBox.critical(None, "Connection", "Connection failed")


if __name__ == "__main__":
    test_dialog()
```

Public API: remote client

class cdl.core.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 cdl.core.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"]
<cdl.core.model.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1]
<cdl.core.model.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1].data
array([1., 2., 3.])
```

connect(*port: str | None = None*, *timeout: float | None = None*, *retries: int | None = None*) → None

Try to connect to DataLab XML-RPC server.

Parameters

- **port** (*str | None*) – XML-RPC port to connect to. If not specified, the port is automatically retrieved from DataLab configuration.

- **timeout** (*float | None*) – Timeout in seconds. Defaults to 5.0.
- **retries** (*int | None*) – Number of retries. Defaults to 10.

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 | None = None*, *yunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*) → bool

Add signal data to DataLab.

Parameters

- **title** (*str*) – Signal title
- **xdata** (*numpy.ndarray*) – X data
- **ydata** (*numpy.ndarray*) – Y data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.

Returns
True if signal was added successfully, False otherwise

Return type
bool

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(*title: str*, *data: ndarray*, *xunit: str | None = None*, *yunit: str | None = None*, *zunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*, *zlabel: str | None = None*) → bool

Add image data to DataLab.

Parameters

- **title** (*str*) – Image title
- **data** (*numpy.ndarray*) – Image data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.

- **zunit** (*str | None*) – Z unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.
- **zlabel** (*str | None*) – Z label. Defaults to None.

Returns
True if image was added successfully, False otherwise

Return type
bool

Raises
ValueError – Invalid data dtype

calc(*name: str*, *param: DataSet | None = None*) → DataSet

Call compute function `name` in current panel's processor.

Parameters

- **name** (*str*) – Compute function name
- **param** (*guidata.dataset.DataSet | None*) – Compute function parameter. Defaults to None.

Returns
Compute function result

Return type
guidata.dataset.DataSet

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** (*list*) – annotation plot items
- **refresh_plot** (*bool | None*) – refresh plot. Defaults to True.
- **panel** (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

add_object(*obj:* SignalObj | ImageObj) → None

Add object to DataLab.

Parameters

obj (SignalObj | ImageObj) – Signal or image object

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** (*str | None*) – Label title. Defaults to None. If None, the title is the object title.
- **panel** (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

close_application() → None

Close DataLab application

context_no_refresh() → Callable

Return a context manager to temporarily disable auto refresh.

Returns

Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.compute_fft()
...     proxy.compute_wiener()
...     proxy.compute_ifft()
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True*, *keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

get_current_panel() → str

Return current panel name.

Returns

Panel name (valid values: "signal", "image", "macro"))

Return type

str

get_group_titles_with_object_infos() → 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 (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

Returns
list of object titles

Return type
list[str]

Raises
ValueError – if panel not found

get_object_uuids(*panel: 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 (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

Returns
list of object uuids

Return type
list[str]

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

Return type
list[str]

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 version.

Returns
DataLab version

Return type
str

import_h5_file(*filename: str*, *reset_all: bool | None = None*) → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- **filename** (*str*) – HDF5 file name
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

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[str] | None*) – List of HDF5 files to open. Defaults to None.
- **import_all** (*bool | None*) – Import all objects from HDF5 files. Defaults to None.
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

open_object(*filename: str*) → None

Open object from file in current panel (signal/image).

Parameters
filename (*str*) – File name

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters
filename (*str*) – 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** (*str | None*) – 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 (*str*) – Panel name (valid values: "signal", "image", "macro"))

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters
state (*bool*) – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters
state (*bool*) – Show titles state

Public API: additional methods

The remote control class methods (either using the proxy or the remote client) may be completed with additional methods which are dynamically added at runtime. This mechanism allows to access the methods of the "processor" objects of DataLab.

Signal Processor

class cdl.core.gui.processor.signal.**SignalProcessor**(*panel: SignalPanel | ImagePanel*, *plotwidget: PlotWidget*)

Object handling signal processing: operations, processing, computing

compute_sum() → None

Compute sum

compute_average() → None

Compute average

compute_product() → None

Compute product

compute_roi_extraction(*param:* ROIDataParam | *None* = *None*) → None

Extract Region Of Interest (ROI) from data

compute_swap_axes() → None

Swap data axes

compute_abs() → None

Compute absolute value

compute_re() → None

Compute real part

compute_im() → None
Compute imaginary part

compute_astype(*param:* DataTypeSParam | *None* = *None*) → None
Convert data type

compute_log10() → None
Compute Log10

compute_difference(*obj2:* SignalObj | *None* = *None*) → None
Compute difference between two signals

compute_quadratic_difference(*obj2:* SignalObj | *None* = *None*) → None
Compute quadratic difference between two signals

compute_division(*obj2:* SignalObj | *None* = *None*) → None
Compute division between two signals

compute_peak_detection(*param:* PeakDetectionParam | *None* = *None*) → None
Detect peaks from data

compute_normalize(*param:* NormalizeYParam | *None* = *None*) → None
Normalize data

compute_derivative() → None
Compute derivative

compute_integral() → None
Compute integral

compute_calibration(*param:* XYCalibrateParam | *None* = *None*) → None
Compute data linear calibration

compute_threshold(*param:* ThresholdParam | *None* = *None*) → None
Compute threshold clipping

compute_clip(*param:* ClipParam | *None* = *None*) → None
Compute maximum data clipping

compute_gaussian_filter(*param:* GaussianParam | *None* = *None*) → None
Compute gaussian filter

compute_moving_average(*param:* MovingAverageParam | *None* = *None*) → None
Compute moving average

compute_moving_median(*param:* MovingMedianParam | *None* = *None*) → None
Compute moving median

compute_wiener() → None
Compute Wiener filter

compute_fft(*param:* FFTParam | *None* = *None*) → None
Compute iFFT

compute_ifft(*param:* FFTParam | *None* = *None*) → None
Compute FFT

compute_interpolation(*obj2:* SignalObj | *None* = *None*, *param:* InterpolationParam | *None* = *None*)
Compute interpolation

compute_resampling(*param:* ResamplingParam | *None* = *None*)
Compute resampling

compute_detrending(*param:* DetrendingParam | *None* = *None*)
Compute detrending

compute_convolution(*obj2:* SignalObj | *None* = *None*) → None
Compute convolution

compute_fit(*name*, *fitdlgfunc*)
Compute fitting curve

compute_polyfit(*param:* PolynomialFitParam | *None* = *None*) → None
Compute polynomial fitting curve

compute_multigaussianfit() → None
Compute multi-Gaussian fitting curve

compute_fwhm(*param:* FWHMParam | *None* = *None*) → dict[str, *ResultShape*]
Compute FWHM

compute_fw1e2() → dict[str, *ResultShape*]
Compute FW at $1/e^2$

Image Processor

class cdl.core.gui.processor.image.**ImageProcessor**(*panel: SignalPanel | ImagePanel*, *plotwidget: PlotWidget*)
Object handling image processing: operations, processing, computing

compute_sum() → None
Compute sum

compute_average() → None
Compute average

compute_product() → None
Compute product

compute_logp1(*param:* LogP1Param | *None* = *None*) → None
Compute base 10 logarithm

compute_rotate(*param:* RotateParam | *None* = *None*) → None
Rotate data arbitrarily

compute_rotate90() → None
Rotate data 90°

compute_rotate270() → None
Rotate data 270°

compute_fliph() → None
Flip data horizontally

compute_flipv() → None
Flip data vertically

distribute_on_grid(*param:* GridParam | *None* = *None*) → None
Distribute images on a grid

reset_positions() → None
Reset image positions

compute_resize(*param:* ResizeParam | *None* = *None*) → None
Resize image

compute_binning(*param:* BinningParam | *None* = *None*) → None
Binning image

compute_roi_extraction(*param:* ROIDataParam | *None* = *None*) → None
Extract Region Of Interest (ROI) from data

compute_profile(*param:* ProfileParam | *None* = *None*) → None
Compute profile

compute_average_profile(*param:* AverageProfileParam | *None* = *None*) → None
Compute average profile

compute_radial_profile(*param:* RadialProfileParam | *None* = *None*) → None
Compute radial profile

compute_swap_axes() → None
Swap data axes

compute_abs() → None
Compute absolute value

compute_re() → None
Compute real part

compute_im() → None
Compute imaginary part

compute_astype(*param:* DataTypeIParam | *None* = *None*) → None
Convert data type

compute_log10() → None
Compute Log10

compute_difference(*obj2:* ImageObj | *None* = *None*) → None
Compute difference between two images

compute_quadratic_difference(*obj2:* ImageObj | *None* = *None*) → None
Compute quadratic difference between two images

compute_division(*obj2:* ImageObj | *None* = *None*) → None
Compute division between two images

compute_flatfield(*obj2:* ImageObj | *None* = *None*, *param:* FlatFieldParam | *None* = *None*) → None
Compute flat field correction

compute_calibration(*param:* ZCalibrateParam | *None* = *None*) → None
Compute data linear calibration

compute_threshold(*param:* ThresholdParam | *None* = *None*) → None
Compute threshold clipping

compute_clip(*param:* ClipParam | *None* = *None*) → None
Compute maximum data clipping

compute_gaussian_filter(*param:* GaussianParam | *None* = *None*) → None
Compute gaussian filter

compute_moving_average(*param:* MovingAverageParam | *None* = *None*) → None
Compute moving average

compute_moving_median(*param:* MovingMedianParam | *None* = *None*) → None
Compute moving median

compute_wiener() → None
Compute Wiener filter

compute_fft(*param:* FFTParam | *None* = *None*) → None
Compute FFT

compute_ifft(*param:* FFTParam | *None* = *None*) → None
Compute iFFT

compute_butterworth(*param:* ButterworthParam | *None* = *None*) → None
Compute Butterworth filter

compute_adjust_gamma(*param:* AdjustGammaParam | *None* = *None*) → None
Compute gamma correction

compute_adjust_log(*param:* AdjustLogParam | *None* = *None*) → None
Compute log correction

compute_adjust_sigmoid(*param:* AdjustSigmoidParam | *None* = *None*) → None
Compute sigmoid correction

compute_rescale_intensity(*param:* RescaleIntensityParam | *None* = *None*) → None
Rescale image intensity levels

compute_equalize_hist(*param:* EqualizeHistParam | *None* = *None*) → None
Histogram equalization

compute_equalize_adapthist(*param:* EqualizeAdaptHistParam | *None* = *None*) → None
Adaptive histogram equalization

compute_denoise_tv(*param:* DenoiseTVParam | *None* = *None*) → None
Compute Total Variation denoising

compute_denoise_bilateral(*param:* DenoiseBilateralParam | *None* = *None*) → None
Compute bilateral filter denoising

compute_denoise_wavelet(*param:* DenoiseWaveletParam | *None* = *None*) → None
Compute Wavelet denoising

compute_denoise_tophat(*param:* MorphologyParam | *None* = *None*) → None
Denoise using White Top-Hat

compute_all_denoise(*params: list* | *None* = *None*) → None
Compute all denoising filters

compute_white_tophat(*param:* MorphologyParam | *None* = *None*) → None
Compute White Top-Hat

compute_black_tophat(*param:* MorphologyParam | *None* = *None*) → None
Compute Black Top-Hat

compute_erosion(*param:* MorphologyParam | *None* = *None*) → None
Compute Erosion

compute_dilation(*param:* MorphologyParam | *None* = *None*) → None
Compute Dilation

compute_opening(*param:* MorphologyParam | *None* = *None*) → None
Compute morphological opening

compute_closing(*param:* MorphologyParam | *None* = *None*) → None
Compute morphological closing

compute_all_morphology(*param:* MorphologyParam | *None* = *None*) → None
Compute all morphology filters

compute_canny(*param:* CannyParam | *None* = *None*) → None
Compute Canny filter

compute_roberts() → None
Compute Roberts filter

compute_prewitt() → None
Compute Prewitt filter

compute_prewitt_h() → None
Compute Prewitt filter (horizontal)

compute_prewitt_v() → None
Compute Prewitt filter (vertical)

compute_sobel() → None
Compute Sobel filter

compute_sobel_h() → None
Compute Sobel filter (horizontal)

compute_sobel_v() → None
Compute Sobel filter (vertical)

compute_scharr() → None
Compute Scharr filter

compute_scharr_h() → None
Compute Scharr filter (horizontal)

compute_scharr_v() → None
Compute Scharr filter (vertical)

compute_farid() → None
Compute Farid filter

compute_farid_h() → None
Compute Farid filter (horizontal)

compute_farid_v() → None
Compute Farid filter (vertical)

compute_laplace() → None
Compute Laplace filter

compute_all_edges() → None
Compute all edges

compute_centroid() → dict[str, *ResultShape*]
Compute image centroid

compute_enclosing_circle() → dict[str, *ResultShape*]
Compute minimum enclosing circle

compute_peak_detection(*param:* Peak2DDetectionParam | *None* = *None*) → dict[str, *ResultShape*]
Compute 2D peak detection

compute_contour_shape(*param:* ContourShapeParam | *None* = *None*) → dict[str, *ResultShape*]
Compute contour shape fit

compute_hough_circle_peaks(*param:* HoughCircleParam | *None* = *None*) → dict[str, *ResultShape*]
Compute peak detection based on a circle Hough transform

compute_blob_dog(*param:* BlobDOGParam | *None* = *None*) → dict[str, *ResultShape*]
Compute blob detection using Difference of Gaussian method

compute_blob_doh(*param:* BlobDOHParam | *None* = *None*) → dict[str, *ResultShape*]
Compute blob detection using Determinant of Hessian method

compute_blob_log(*param:* BlobLOGParam | *None* = *None*) → dict[str, *ResultShape*]
Compute blob detection using Laplacian of Gaussian method

compute_blob_opencv(*param:* BlobOpenCVParam | *None* = *None*) → dict[str, *ResultShape*]
Compute blob detection using OpenCV

2.1.4 Internal data model

In its internal data model, DataLab stores data using two main classes:

- *cdl.obj.SignalObj*, which represents a signal object, and
- *cdl.obj.ImageObj*, which represents an image object.

These classes are defined in the `cdl.core.model` package but are exposed publicly in the *cdl.obj* 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 *cdl.param* package.

See also:

The *API* section for more information on the public API.

2.1.5 Plugins

DataLab is a modular application. It is possible to add new features to DataLab by writing plugins. A plugin is a Python module that is loaded at startup by DataLab. A plugin may add new features to DataLab, or modify existing features.

The plugin system currently supports the following features:

- Processing features: add new processing tasks to the DataLab processing system, including specific graphical user interfaces.
- Input/output features: add new file formats to the DataLab file I/O system.
- HDF5 features: add new HDF5 file formats to the DataLab HDF5 I/O system.

What is a plugin?

A plugin is a Python module that is loaded at startup by DataLab. A plugin may add new features to DataLab, or modify existing features.

A plugin is a Python module that contains a class derived from the *cdl.plugins.PluginBase* class. The name of the class is not important, as long as it is derived from *cdl.plugins.PluginBase* and has a `PLUGIN_INFO` attribute that is an instance of the *cdl.plugins.PluginInfo* class. The `PLUGIN_INFO` attribute is used by DataLab to retrieve information about the plugin.

Where to put a plugin?

As plugins are Python modules, they can be put anywhere in the Python path of the DataLab installation.

Special additional locations are available for plugins:

- The *plugins* directory in the user configuration folder (e.g. *C:\Users\JohnDoe\.DataLab\plugins* on Windows or *~/.DataLab/plugins* on Linux).
- The *plugins* directory in the same folder as the *DataLab* executable in case of a standalone installation.
- The *plugins* directory in the *cdl* package in case for internal plugins only (i.e. it is not recommended to put your own plugins there).

Example: processing plugin

Here is a simple example of a plugin that adds a new features to DataLab.

```
# -*- coding: utf-8 -*-
#
# Licensed under the terms of the BSD 3-Clause
# (see cdl/LICENSE for details)

"""
Test Data Plugin for DataLab
----------------------------

This plugin is an example of DataLab plugin. It provides test data samples
and some actions to test DataLab functionalities.
"""

import cdl.obj as dlo
```

(continues on next page)

(continued from previous page)

```
import cdl.tests.data as test_data
from cdl.config import _
from cdl.core.computation import image as cpima
from cdl.core.computation import signal as cpsig
from cdl.plugins import PluginBase, PluginInfo

# ------------------------------------------------------------------------------
# All computation functions must be defined as global functions, otherwise
# they cannot be pickled and sent to the worker process
# ------------------------------------------------------------------------------


def add_noise_to_signal(
    src: dlo.SignalObj, p: test_data.GaussianNoiseParam
) -> dlo.SignalObj:
    """Add gaussian noise to signal"""
    dst = cpsig.dst_11(src, "add_gaussian_noise", f"mu={p.mu},sigma={p.sigma}")
    test_data.add_gaussian_noise_to_signal(dst, p)
    return dst


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"),
    )

    # Signal processing features ----------------------------------------------
    def add_noise_to_signal(self) -> None:
        """Add noise to signal"""
        self.signalpanel.processor.compute_11(
            add_noise_to_signal,
            paramclass=test_data.GaussianNoiseParam,
            title=_("Add noise"),
        )

    def create_paracetamol_signal(self) -> None:
        """Create paracetamol signal"""
        obj = test_data.create_paracetamol_signal()
        self.proxy.add_object(obj)

    def create_noisy_signal(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        """Create noisy signal"""
        obj = self.signalpanel.new_object(add_to_panel=False)
        if obj is not None:
            noiseparam = test_data.GaussianNoiseParam(_("Noise"))
            self.signalpanel.processor.update_param_defaults(noiseparam)
            if noiseparam.edit(self.signalpanel):
                test_data.add_gaussian_noise_to_signal(obj, noiseparam)
                self.proxy.add_object(obj)

    # Image processing features ------------------------------------------------
    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_peak2d_image(self) -> None:
        """Create 2D peak image"""
        obj = self.imagepanel.new_object(add_to_panel=False)
        param = test_data.PeakDataParam.create(size=max(obj.data.shape))
        self.imagepanel.processor.update_param_defaults(param)
        if param.edit(self.imagepanel):
            obj.data = test_data.get_peak2d_data(param)
            self.proxy.add_object(obj)

    def create_sincos_image(self) -> None:
        """Create 2D sin cos image"""
        newparam = self.edit_new_image_parameters(hide_image_type=True)
        if newparam is not None:
            obj = test_data.create_sincos_image(newparam)
            self.proxy.add_object(obj)

    def create_noisygauss_image(self) -> None:
        """Create 2D noisy gauss image"""
        newparam = self.edit_new_image_parameters(hide_image_type=True)
        if newparam is not None:
            obj = test_data.create_noisygauss_image(newparam)
            self.proxy.add_object(obj)

    def create_multigauss_image(self) -> None:
        """Create 2D multi gauss image"""
        newparam = self.edit_new_image_parameters(hide_image_type=True)
        if newparam is not None:
            obj = test_data.create_multigauss_image(newparam)
            self.proxy.add_object(obj)

    def create_2dstep_image(self) -> None:
        """Create 2D step image"""
        newparam = self.edit_new_image_parameters(hide_image_type=True)
        if newparam is not None:
```

(continues on next page)

(continued from previous page)

```
            obj = test_data.create_2dstep_image(newparam)
            self.proxy.add_object(obj)

    def create_ring_image(self) -> None:
        """Create 2D ring image"""
        param = test_data.RingParam(_("Ring"))
        if param.edit(self.imagepanel):
            obj = test_data.create_ring_image(param)
            self.proxy.add_object(obj)

    def create_annotated_image(self) -> None:
        """Create annotated image"""
        obj = test_data.create_annotated_image()
        self.proxy.add_object(obj)

    # Plugin menu entries ------------------------------------------------------
    def create_actions(self) -> None:
        """Create actions"""
        # Signal Panel ---------------------------------------------------------
        sah = self.signalpanel.acthandler
        with sah.new_menu(_("Test data")):
            sah.new_action(_("Add noise to signal"), triggered=self.add_noise_to_signal)
            sah.new_action(
                _("Load spectrum of paracetamol"),
                triggered=self.create_paracetamol_signal,
                select_condition="always",
                separator=True,
            )
            sah.new_action(
                _("Create noisy signal"),
                triggered=self.create_noisy_signal,
                select_condition="always",
            )
        # Image Panel ----------------------------------------------------------
        iah = self.imagepanel.acthandler
        with iah.new_menu(_("Test data")):
            iah.new_action(_("Add noise to image"), triggered=self.add_noise_to_image)
            # with iah.new_menu(_("Data samples")):
            iah.new_action(
                _("Create image with peaks"),
                triggered=self.create_peak2d_image,
                select_condition="always",
                separator=True,
            )
            iah.new_action(
                _("Create 2D sin cos image"),
                triggered=self.create_sincos_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D noisy gauss image"),
                triggered=self.create_noisygauss_image,
```

(continues on next page)

(continued from previous page)

```
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D multi gauss image"),
                triggered=self.create_multigauss_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create annotated image"),
                triggered=self.create_annotated_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create 2D step image"),
                triggered=self.create_2dstep_image,
                select_condition="always",
            )
            iah.new_action(
                _("Create ring image"),
                triggered=self.create_ring_image,
                select_condition="always",
            )
```

Example: input/output plugin

Here is a simple example of a plugin that adds a new file formats to DataLab.

```
# -*- coding: utf-8 -*-
#
# Licensed under the terms of the BSD 3-Clause
# (see cdl/LICENSE for details)

"""
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 cdl.core.io.base import FormatInfo
from cdl.core.io.image.base import ImageFormatBase

# ===============================================================================
# Thales Pixium FXD file format
# ===============================================================================
```

(continues on next page)

(continued from previous page)

```
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
        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:
```

(continues on next page)

(continued from previous page)

```
            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)
            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.fromstring(block, dtype=dtype)
        self.data = data.reshape(self.nbrows, self.nbcols)


class FXDImageFormat(ImageFormatBase):
    """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:
```

(continues on next page)

(continued from previous page)

```
            np.ndarray: image data
        """
        fxd_file = FXDFile(filename)
        return fxd_file.data


# ============================================================================
# Dürr NDT XYZ file format
# ============================================================================


class XYZImageFormat(ImageFormatBase):
    """Object representing Dürr NDT XYZ image file type"""

    FORMAT_INFO = FormatInfo(
        name="Dürr NDT",
        extensions="*.xyz",
        readable=True,
        writeable=True,
    )

    @staticmethod
    def read_data(filename: str) -> np.ndarray:
        """Read data and return it

        Args:
            filename (str): path to XYZ file

        Returns:
            np.ndarray: image data
        """
        with open(filename, "rb") as fdesc:
            cols = int(np.fromfile(fdesc, dtype=np.uint16, count=1)[0])
            rows = int(np.fromfile(fdesc, dtype=np.uint16, count=1)[0])
            arr = np.fromfile(fdesc, dtype=np.uint16, count=cols * rows)
            arr = arr.reshape((rows, cols))
        return np.fliplr(arr)

    @staticmethod
    def write_data(filename: str, data: np.ndarray) -> None:
        """Write data to file

        Args:
            filename: File name
            data: Image array data
        """
        data = np.fliplr(data)
        with open(filename, "wb") as fdesc:
            fdesc.write(np.array(data.shape[1], dtype=np.uint16).tobytes())
            fdesc.write(np.array(data.shape[0], dtype=np.uint16).tobytes())
            fdesc.write(data.tobytes())
```

Other examples

Other examples of plugins can be found in the *plugins/examples* directory of the DataLab source code (explore here on GitHub).

Public API

DataLab plugin system

DataLab plugin system provides a way to extend the application with new functionalities.

Plugins are Python modules that relies on two classes:

- `PluginInfo`, which stores information about the plugin
- `PluginBase`, which is the base class for all plugins

Plugins may also extends DataLab I/O features by providing new image or signal formats. To do so, they must provide a subclass of `ImageFormatBase` or `SignalFormatBase`, in which format infos are defined using the `FormatInfo` class.

class cdl.plugins.**PluginRegistry**(*name*, *bases*, *attrs*)

Metaclass for registering plugins

classmethod get_plugin_classes() → list[*PluginBase*]

Return plugin classes

classmethod get_plugins() → list[*PluginBase*]

Return plugin instances

classmethod get_plugin(*name_or_class*) → *PluginBase* | None

Return plugin instance

classmethod register_plugin(*plugin:* PluginBase)

Register plugin

classmethod unregister_plugin(*plugin:* PluginBase)

Unregister plugin

classmethod get_plugin_infos() → str

Return plugin infos (names, versions, descriptions) in html format

class cdl.plugins.**PluginInfo**(*name: str = None*, *version: str = '0.0.0'*, *description: str = ''*, *icon: str = None*)

Plugin info

class cdl.plugins.**PluginBaseMeta**(*name*, *bases*, *namespace*, */*, ***kwargs*)

Mixed metaclass to avoid conflicts

class cdl.plugins.**PluginBase**

Plugin base class

property signalpanel: SignalPanel

Return signal panel

property imagepanel: ImagePanel

Return image panel

show_warning(*message: str*)

Show warning message

show_error(*message: str*)

Show error message

show_info(*message: str*)

Show info message

ask_yesno(*message: str*, *title: str | None = None*, *cancelable: bool = False*) → bool

Ask yes/no question

edit_new_signal_parameters(*title: str | None = None*, *size: int | None = None*, *hide_signal_type: bool = True*) → *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)
- **hide_signal_type** – hide signal type parameter (default: True)

Returns

New signal parameter dataset (or None if canceled)

edit_new_image_parameters(*title: str | None = None*, *shape: tuple[int, int] | None = None*, *hide_image_type: bool = True*, *hide_image_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_image_type** – hide image type parameter (default: True)
- **hide_image_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.CDLMainWindow*) → None

Register plugin

unregister()

Unregister plugin

register_hooks()

Register plugin hooks

unregister_hooks()

Unregister plugin hooks

abstract create_actions()

Create actions

cdl.plugins.**discover_plugins**() → list[*PluginBase*]

Discover plugins using naming convention

2.1.6 Log viewer

Despite countless efforts (unit testing, test coverage, ...), DataLab might crash or behave unexpectedly.

For those situations, DataLab provides two logs (located in your home directory):

- "Traceback log", for Python exceptions
- "Faulthandler log", for system failures (e.g. Qt-related crash)

Fig. 6: DataLab log viewer (see "?" menu)

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 viewer*).

2.1.7 Installation and configuration viewer

Because of the multiple ways of installing DataLab on your machine, understanding why the application behaves unexpectedly without any information on your configuration could be very challenging.

That is why DataLab provides the dialog box "Installation and configuration" which gathers all the information about your installation and configuration.

Reporting unexpected behavior or any other bug on GitHub Issues will be greatly appreciated, especially if above contents are attached to the report (as well log files, see *Log viewer*).

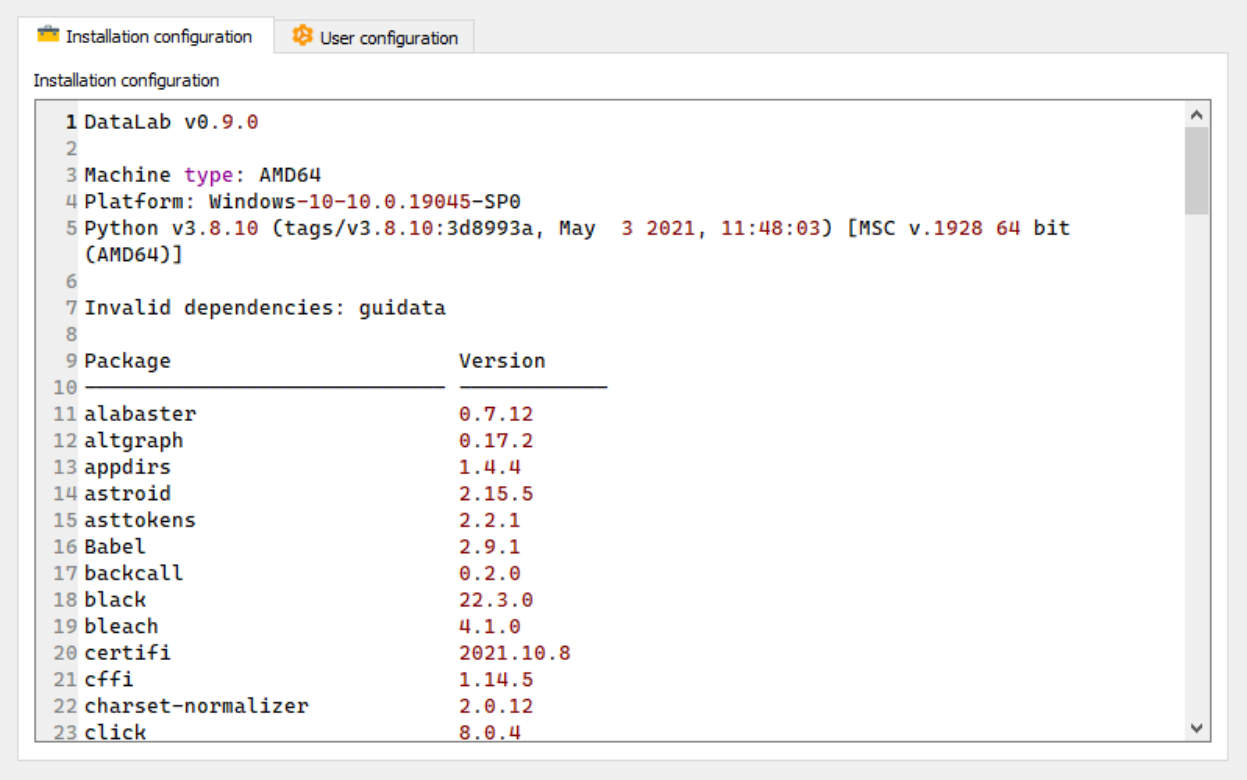


Fig. 7: Installation and configuration (see "?" menu)

2.1.8 Command line features

Run DataLab

To run DataLab from the command line, type the following:

```
$ cdl
```

To show help on command line usage, simply run:

```
$ cdl --help
usage: app.py [-h] [-b path] [-v] [--unattended] [--screenshot] [--delay DELAY] [--
↪xmlrpcport PORT]
              [--verbose {quiet,minimal,normal}]
              [h5]

Run DataLab

positional arguments:
  h5                    HDF5 file names (separated by ';'), optionally with dataset name␣
↪(separated by ',')

optional arguments:
  -h, --help            show this help message and exit
  -b path, --h5browser path
```

(continues on next page)

(continued from previous page)

```
                        path to open with HDF5 browser
  -v, --version         show DataLab version
  --unattended          non-interactive mode
  --screenshot          automatic screenshots
  --delay DELAY         delay (seconds) before quitting application in unattended mode
  --xmlrpcport XMLRPCPORT
                        XML-RPC port number
  --verbose {quiet,minimal,normal}
                        verbosity level: for debugging/testing purpose
```

Open HDF5 file at startup

To open HDF5 files, or even import only a specified HDF5 dataset, use the following:

```
$ cdl /path/to/file1.h5
$ cdl /path/to/file1.h5,/path/to/dataset1
$ cdl /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:

```
$ cdl -b /path/to/file1.h5
$ cdl --h5browser /path/to/file1.h5
```

Run DataLab demo

To execute DataLab demo, run the following:

```
$ cdl-demo
```

Run unit tests

Note: This test suite is based on *guidata.guitest* discovery mechanism. It is not compatible with *pytest* because most of the high level tests have to be executed in a separate process (e.g. scenario tests will fail if executed in the same process as other tests).

To execute all DataLab unit tests, simply run:

```
$ cdl-alltests

================================================================================
 DataLab v0.9.0 automatic unit tests
================================================================================

 DataLab characteristics/environment:
```

(continues on next page)

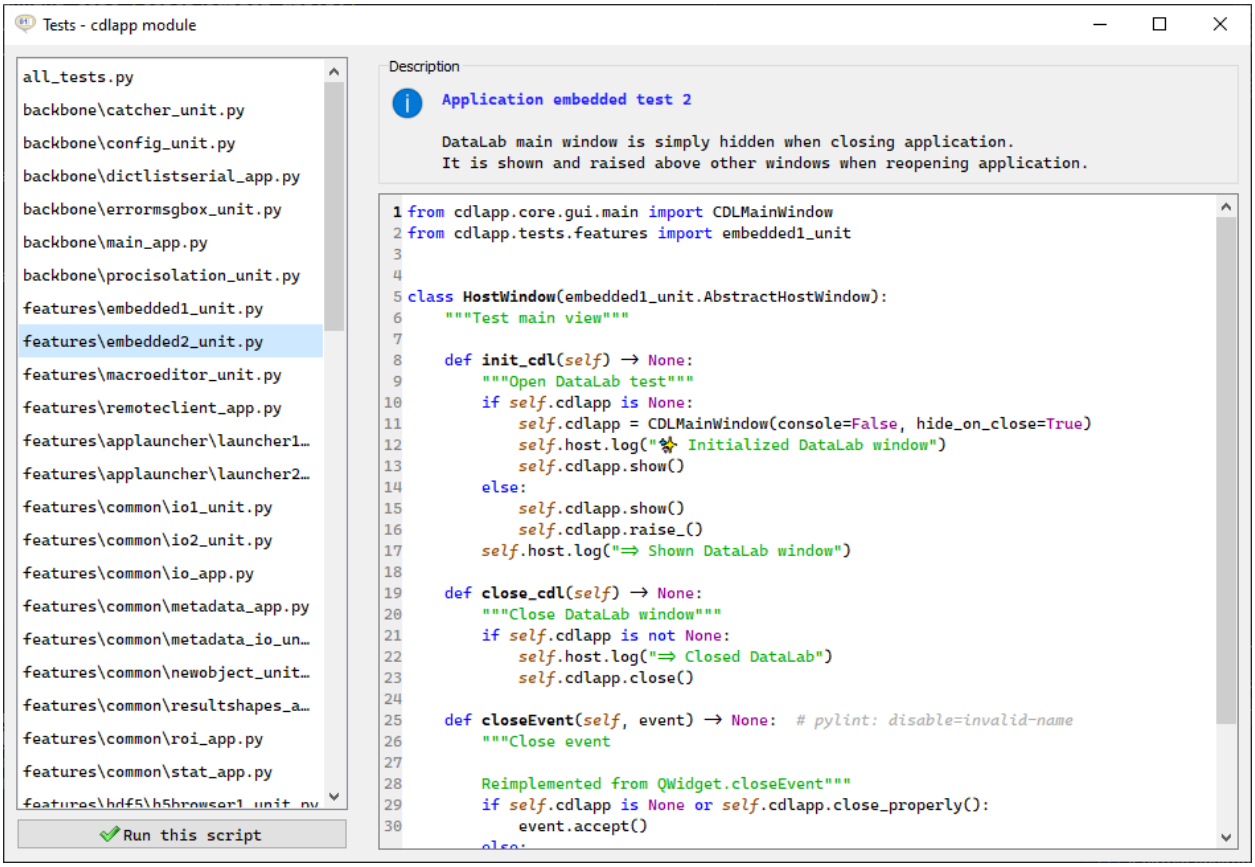
(continued from previous page)

```
Configuration version: 1.0.0
Path: C:\Dev\Projets\DataLab\cdl
Frozen: False
Debug: False

 DataLab configuration:
Process isolation: enabled
RPC server: enabled
Console: enabled
Available memory threshold: 500 MB
Ignored dependencies: disabled
Processing:
    Extract all ROIs in a single signal or image
    FFT shift: enabled

 Test parameters:
 Selected 51 tests (51 total available)
 Test data path:
    C:\Dev\Projets\DataLab\cdl\data\tests
 Environment:
    CDL_DATA=C:\Dev\Projets\DataLab_data\
    PYTHONPATH=.
    DEBUG=

Please wait while test scripts are executed (a few minutes).
Only error messages will be printed out (no message = test OK).

===[01/51]===  Running test [tests\annotations_app.py]
===[02/51]===  Running test [tests\annotations_unit.py]
===[03/51]===  Running test [tests\auto_app.py]
===[04/51]===  Running test [tests\basic1_app.py]
===[05/51]===  Running test [tests\basic2_app.py]
===[06/51]===  Running test [tests\basic3_app.py]
```

Run interactive tests

To execute DataLab interactive tests, run the following:

```
$ cdl-tests
```


Tests - cdlapp module
all_tests.py
backbone\catcher_unit.py
backbone\config_unit.py
backbone\dictlistserial_app.py
backbone\errormsgbox_unit.py
backbone\main_app.py
backbone\procisolation_unit.py
features\embedded1_unit.py
features\embedded2_unit.py
features\macroeditor_unit.py
features\remoteclient_app.py
features\applauncher\launcher1…
features\applauncher\launcher2…
features\common\io1_unit.py
features\common\io2_unit.py
features\common\io_app.py
features\common\metadata_app.py
features\common\metadata_io_un…
features\common\newobject_unit…
features\common\resultshapes_a…
features\common\roi_app.py
features\common\stat_app.py
Run this script
Description
Application embedded test 2
DataLab main window is simply hidden when closing application.
It is shown and raised above other windows when reopening application.
from cdlapp.core.gui.main import CDLMainWindow
from cdlapp.tests.features import embedded1_unit
class HostWindow(embedded1_unit.AbstractHostWindow):
"""Test main view"""
def init_cdl(self) → None:
"""Open DataLab test"""
if self.cdlapp is None:
self.cdlapp = CDLMainWindow(console=False, hide_on_close=True)
self.host.log("✨ Initialized DataLab window")
self.cdlapp.show()
else:
self.cdlapp.show()
self.cdlapp.raise_()
self.host.log("⇒ Shown DataLab window")
def close_cdl(self) → None:
"""Close DataLab window"""
if self.cdlapp is not None:
self.host.log("⇒ Closed DataLab")
self.cdlapp.close()
def closeEvent(self, event) → None: # pylint: disable=invalid-name
"""Close event
Reimplemented from QWidget.closeEvent"""
if self.cdlapp is None or self.cdlapp.close_properly():
event.accept()

2.2 Signal processing

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

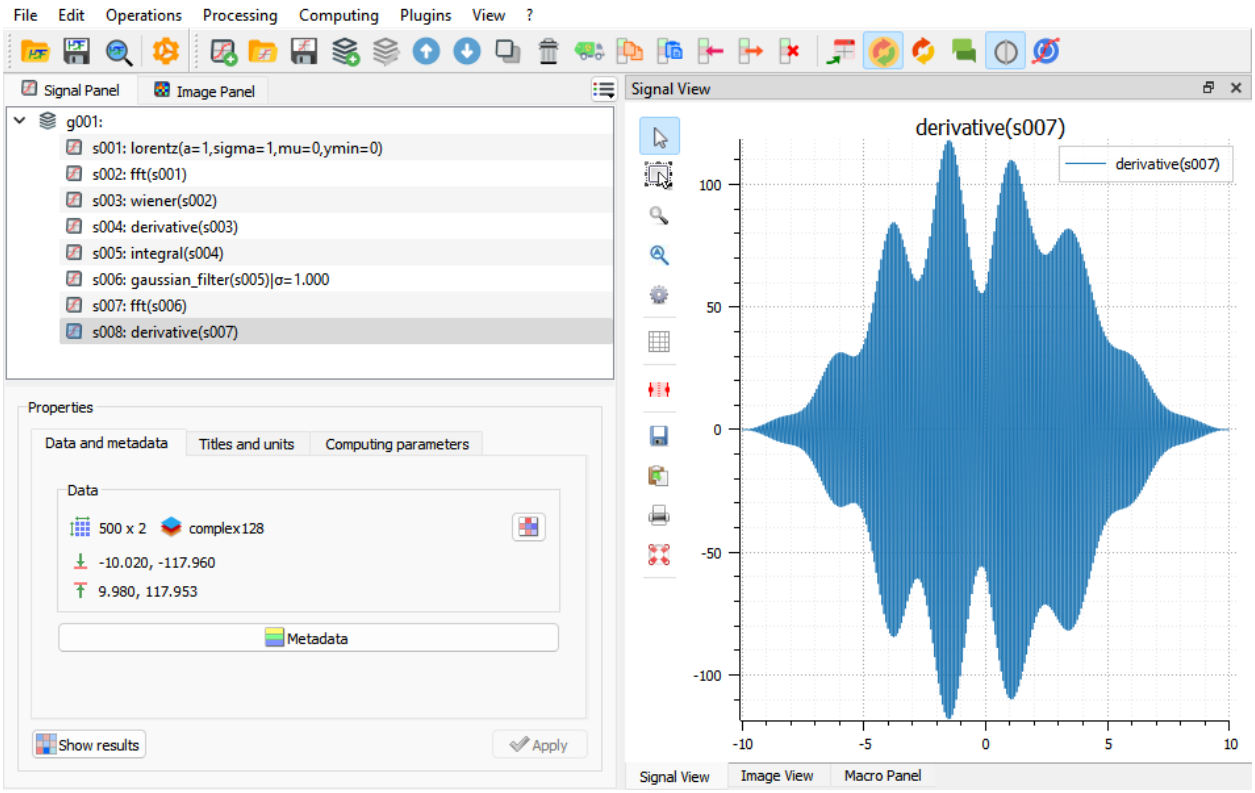


Fig. 8: DataLab main window: Signal processing view

2.2.1 Menus

This section describes the signal related feature of DataLab, by presenting the different menus and their entries.

"File" menu

The "File" menu allows you to create, open, save and close signals. It also allows you to import and export data from/to HDF5 files, and to edit the settings of the current session.

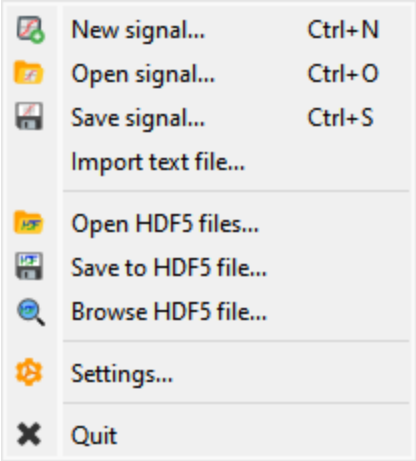


Fig. 9: Screenshot of the "File" menu.

New signal

Create a new signal from various models:

Model	Equation
Zeros	$y[i] = 0$
Random	$y[i] \in [-0.5, 0.5]$
Gaussian	$y = y_0 + \frac{A}{\sqrt{2\pi}.\sigma}.exp(-\frac{1}{2}.(\frac{x-x_0}{\sigma})^2)$
Lorentzian	$y = y_0 + \frac{A}{\sigma.\pi}.\frac{1}{1+(\frac{x-x_0}{\sigma})^2}$
Voigt	$y = y_0 + A.\frac{Re(exp(-z^2).erfc(-j.z))}{\sqrt{2\pi}.\sigma}$ with $z = \frac{x-x_0-j.\sigma}{\sqrt{2}.\sigma}$

Open signal

Create a new signal from the following supported filetypes:

File type	Extensions
Text files	.txt, .csv
NumPy arrays	.npy

Save signal

Save current signal to the following supported filetypes:

File type	Extensions
Text files	.csv

Import text file

Import data from a text file.

See also:

See *Signal Text File Import* page for more details on importing text files.

Open HDF5 file

Import data from a HDF5 file.

Save to HDF5 file

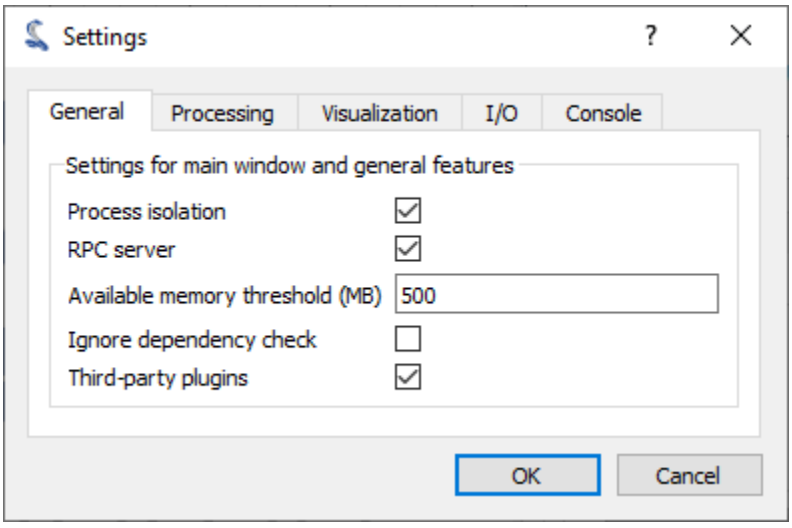
Export the whole DataLab session (all signals and images) into a HDF5 file.

Browse HDF5 file

Open the *HDF5 Browser* in a new window to browse and import data from HDF5 file.

Settings

Open the the "Settings" dialog box.



"Edit" menu

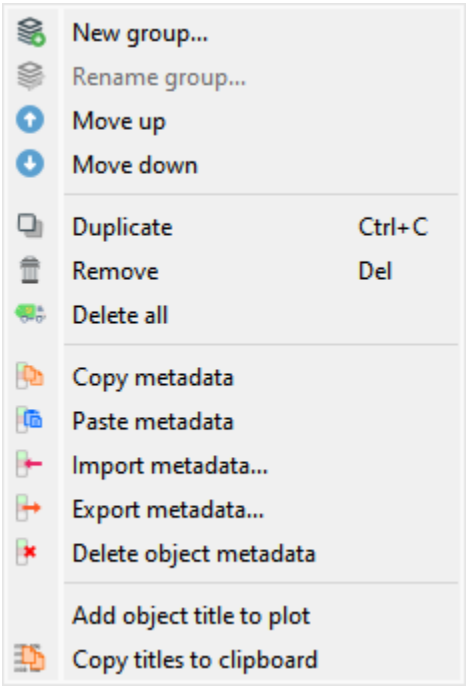


Fig. 10: Screenshot of the "Edit" menu.

The "Edit" menu allows you to edit the current signal or group of signals, by adding, removing, renaming, moving up or down, or duplicating signals. It also manipulates metadata, or handles signal titles.

New group

Create a new group of signals. Images may be moved from one group to another by drag and drop.

Rename group

Rename currently selected group.

Move up

Move current selection up in the list (groups or signals may be selected). If multiple objects are selected, they are moved together. If a selected signal is already at the top of its group, it is moved to the bottom of the previous group.

Move down

Move current selection down in the list (groups or signals may be selected). If multiple objects are selected, they are moved together. If a selected signal is already at the bottom of its group, it is moved to the top of the next group.

Duplicate

Create a new signal which is identical to the currently selected object.

Remove

Remove currently selected signal.

Delete all

Delete all signals.

Copy metadata

Copy metadata from currently selected signal into clipboard.

Paste metadata

Paste metadata from clipboard into selected signal.

Import metadata into signal

Import metadata from a JSON text file.

Export metadata from signal

Export metadata to a JSON text file.

Delete object metadata

Delete metadata from currently selected signal. Metadata contains additionnal information such as Region of Interest or results of computations

Add object title to plot

Add currently selected signal title to the associated plot.

Copy titles to clipboard

Copy all signal titles to clipboard as a multiline text. This text may be used for reproducing a processing chain, for example.

"Operation" menu

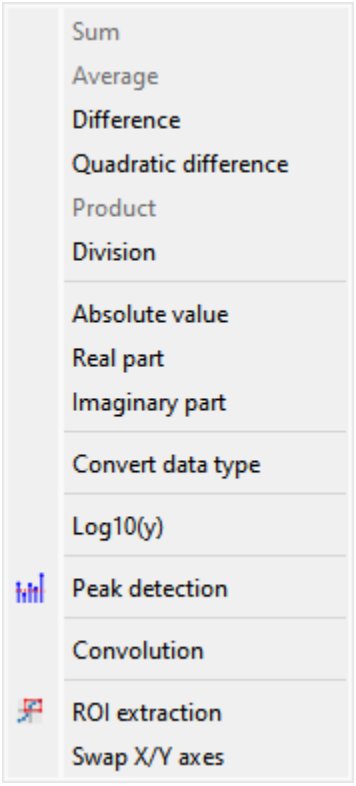


Fig. 11: Screenshot of the "Operation" menu.

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

Sum

Create a new signal which is the sum of all selected signals:

$$y_M = \sum_{k=0}^{M-1} y_k$$

Average

Create a new signal which is the average of all selected signals:

$$y_M = \frac{1}{M} \sum_{k=0}^{M-1} y_k$$

Difference

Create a new signal which is the difference of the **two** selected signals:

$$y_2 = y_1 - y_0$$

Product

Create a new signal which is the product of all selected signals:

$$y_M = \prod_{k=0}^{M-1} y_k$$

Division

Create a new signal which is the division of the **two** selected signals:

$$y_2 = \frac{y_1}{y_0}$$

Absolute value

Create a new signal which is the absolute value of each selected signal:

$$y_k = |y_{k-1}|$$

Real part

Create a new signal which is the real part of each selected signal:

$$y_k = \Re(y_{k-1})$$

Imaginary part

Create a new signal which is the imaginary part of each selected signal:

$$y_k = \Im(y_{k-1})$$

Convert data type

Create a new signal which is the result of converting data type of each selected signal.

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

Log10(y)

Create a new signal which is the base 10 logarithm of each selected signal:

$$z_k = \log_{10}(z_{k-1})$$

Peak detection

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

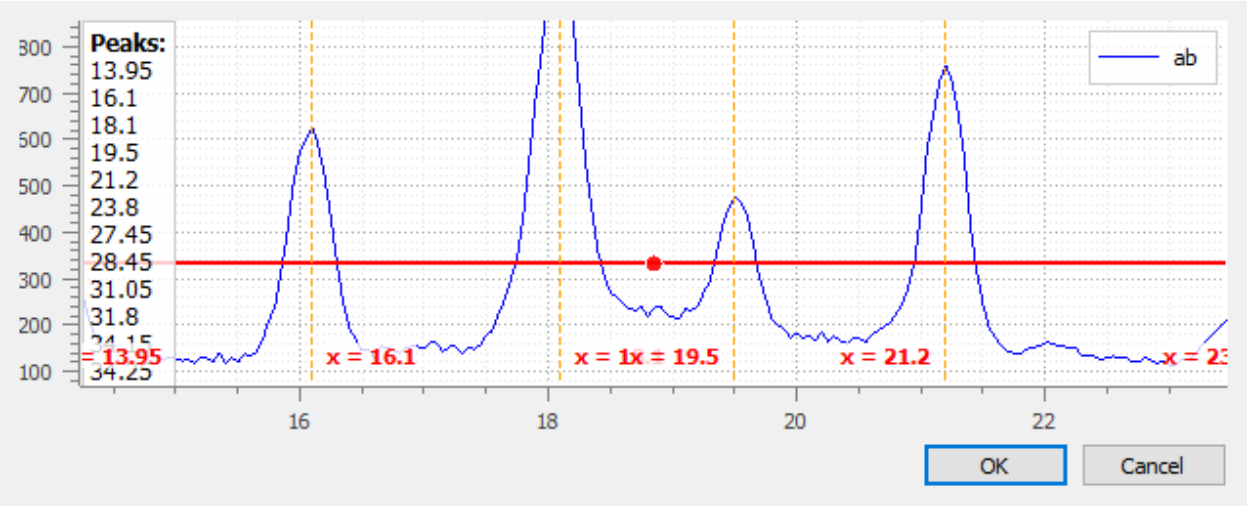


Fig. 12: Peak detection dialog: threshold is adjustable by moving the horizontal marker, peaks are detected automatically (see vertical markers with labels indicating peak position)

Convolution

Create a new signal which is the convolution of each selected signal with respect to another signal.

This feature is based on SciPy's scipy.signal.convolve function.

ROI extraction

Create a new signal from a user-defined Region of Interest (ROI).

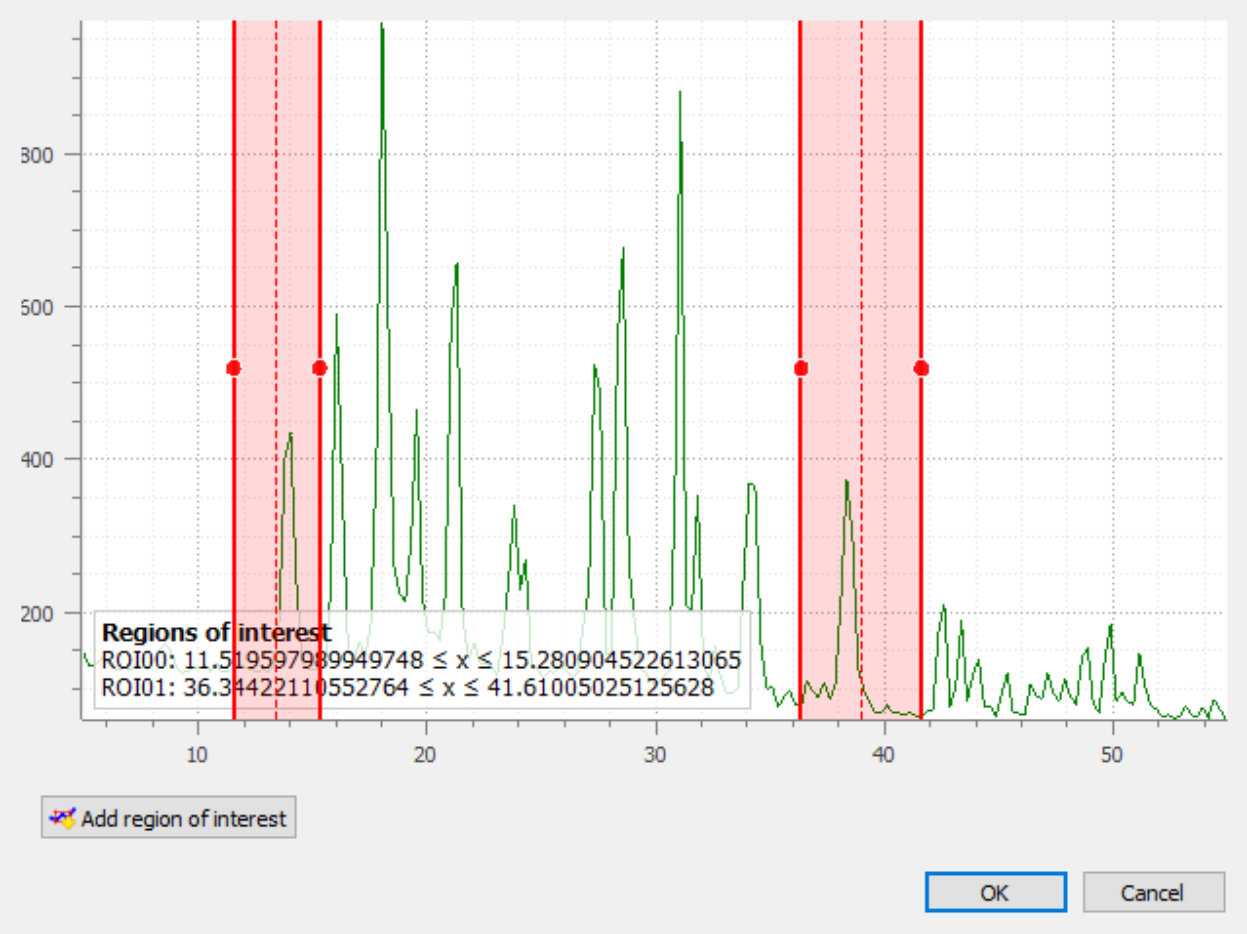


Fig. 13: ROI extraction dialog: the ROI is defined by moving the position and adjusting the width of an horizontal range.

Swap X/Y axes

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

"Processing" menu

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

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

Normalize

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

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

Derivative

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

Integral

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

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	$y_1 = a.y_0 + b$

Gaussian filter

Compute 1D-Gaussian filter of each selected signal (implementation based on scipy.ndimage.gaussian_filter1d).

Moving average

Compute moving average on M points of each selected signal, without border effect:

$$y_1[i] = \frac{1}{M} \sum_{j=0}^{M-1} y_0[i+j]$$

Moving median

Compute moving median of each selected signal (implementation based on scipy.signal.medfilt).

Wiener filter

Compute Wiener filter of each selected signal (implementation based on scipy.signal.wiener).

FFT

Create a new signal which is the Fast Fourier Transform (FFT) of each selected signal.

Inverse FFT

Create a new signal which is the inverse FFT of each selected signal.

Interpolation

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

The following interpolation methods are available:

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

Resampling

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

The following parameters are available:

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

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.

Lorentzian, Voigt, Polynomial and Multi-Gaussian fit

Open an interactive curve fitting tool in a modal dialog box.

Model	Equation
Gaussian	$y = y_0 + \frac{A}{\sqrt{2\pi}.\sigma}.exp(-\frac{1}{2}.(\frac{x-x_0}{\sigma})^2)$
Lorentzian	$y = y_0 + \frac{A}{\sigma.\pi}.\frac{1}{1+(\frac{x-x_0}{\sigma})^2}$
Voigt	$y = y_0 + A.\frac{Re(exp(-z^2).erfc(-j.z))}{\sqrt{2\pi}.\sigma}$ with $z = \frac{x-x_0-j.\sigma}{\sqrt{2}.\sigma}$
Multi-Gaussian	$y = y_0 + \sum_{i=0}^{K} \frac{A_i}{\sqrt{2\pi}.\sigma_i}.exp(-\frac{1}{2}.(\frac{x-x_{0,i}}{\sigma_i})^2)$

"Computing" menu

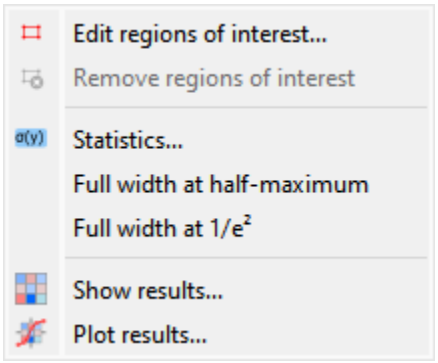


Fig. 15: Screenshot of the "Computing" menu.

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

Note: In DataLab vocabulary, a "computing" 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.

Edit regions of interest

Open a dialog box to setup multiple Region Of Interests (ROI). ROI are stored as metadata, and thus attached to signal.

ROI definition dialog is exactly the same as ROI extraction (see above): the ROI is defined by moving the position and adjusting the width of an horizontal range.

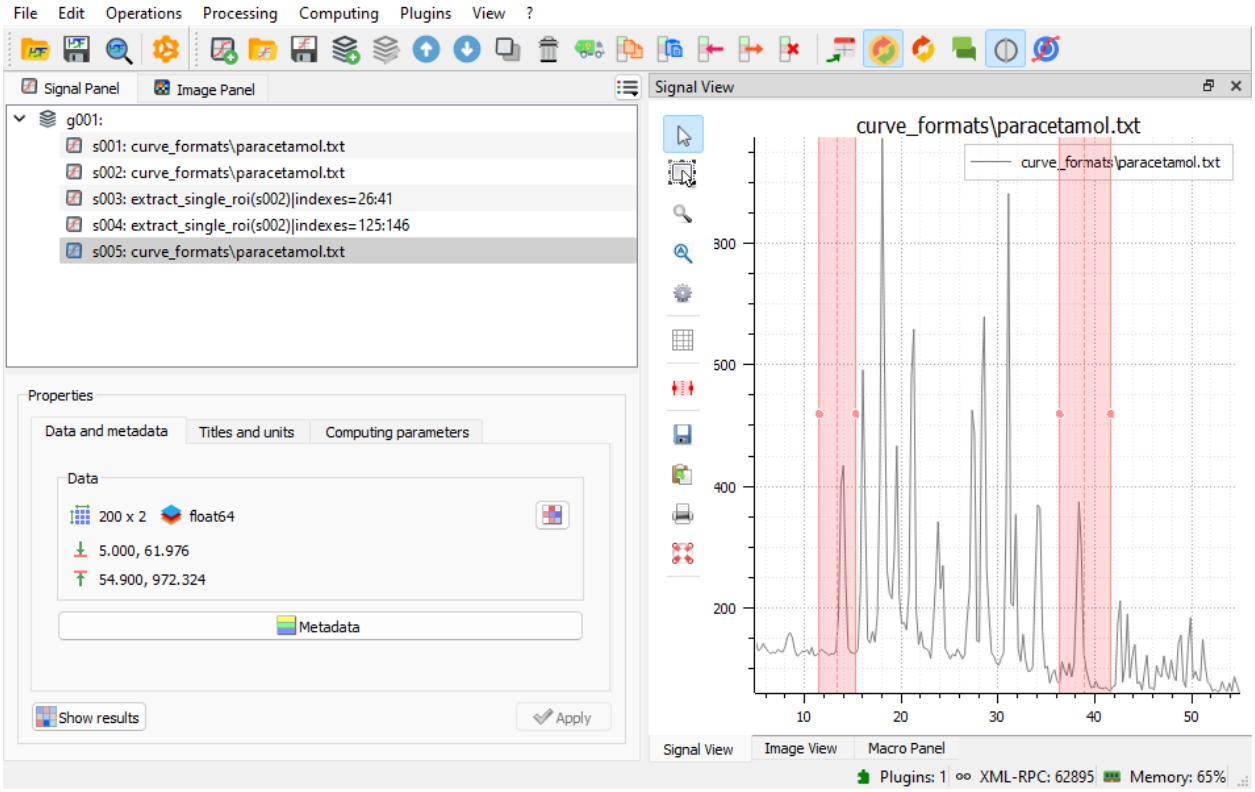


Fig. 16: A signal with an ROI.

Remove regions of interest

Remove all defined ROI for selected object(s).

Statistics

Compute statistics on selected signal and show a summary table.

Full width at half-maximum

Fit data to a Gaussian, Lorentzian or Voigt model using least-square method. Then, compute the full width at half-maximum value.

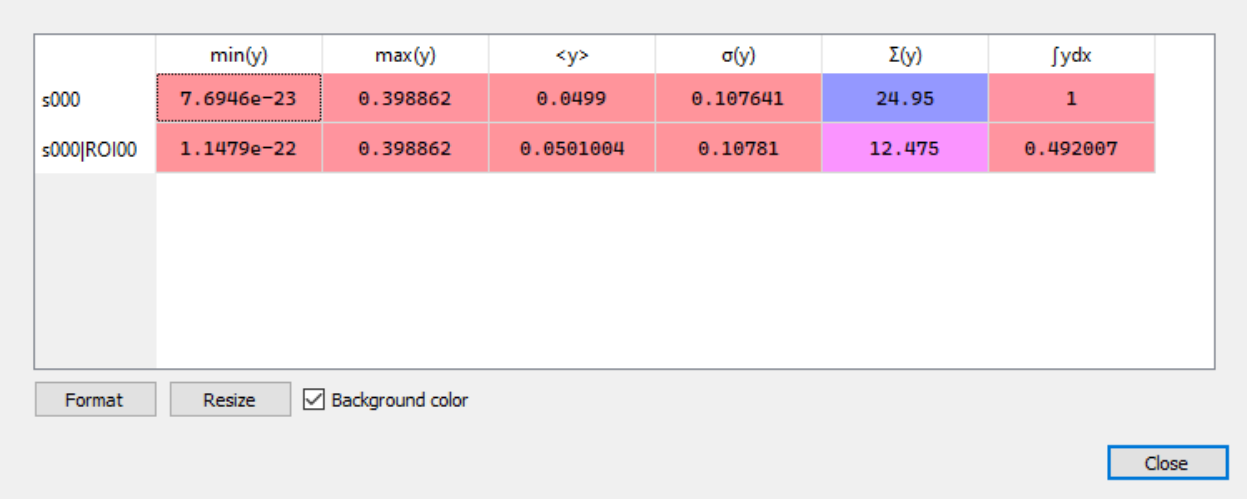


Fig. 17: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

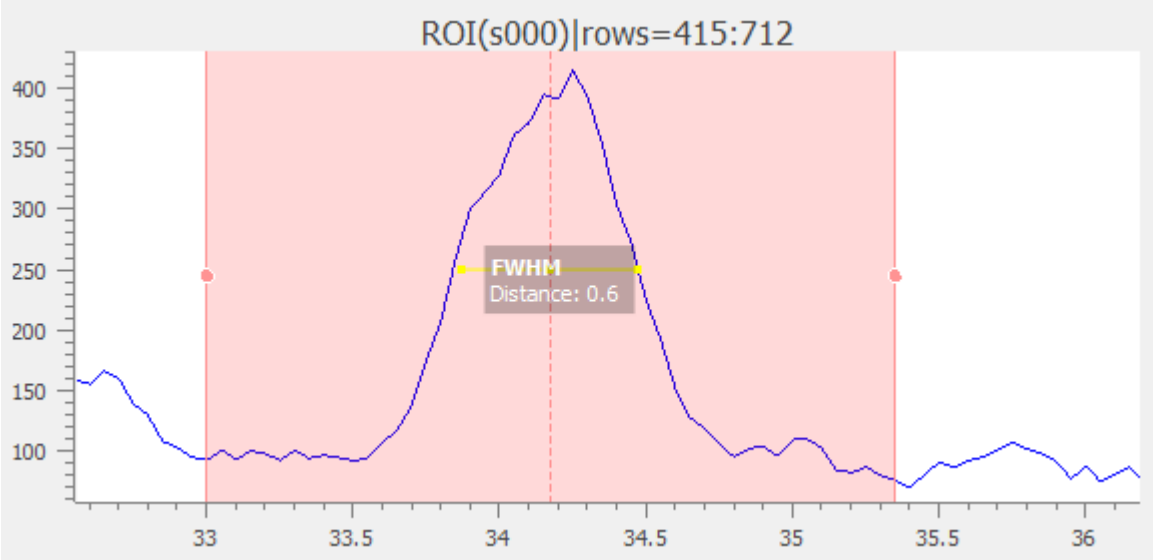


Fig. 18: The computed result is displayed as an annotated segment.

Full width at $1/e^2$

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

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

Show results

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

Plot results

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

"View" menu

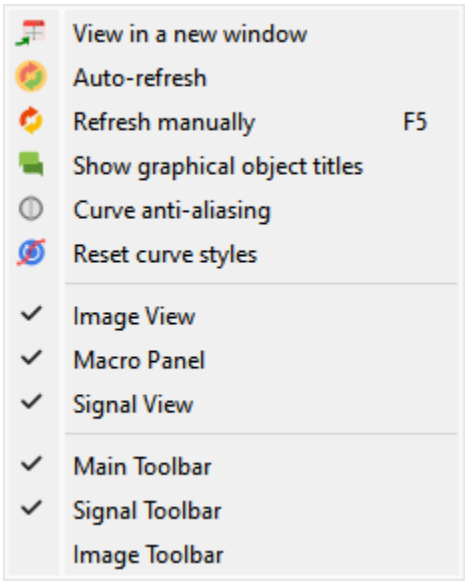


Fig. 19: Screenshot of the "View" menu.

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 and the selected signals.

In the separate window, you may visualize your data more comfortably (e.g., by maximizing the window) and you may also annotate the data.

See also:

See *Annotations (Signals)* for more details on annotations.

Show graphical object titles

Show/hide titles of computing results or 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 clicking the "Refresh manually" button in the toolbar. 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.

Refresh manually

Refresh the visualization manually. This triggers a refresh of the plot view, even if the auto-refresh feature is disabled.

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

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

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

Other menu entries

Show/hide panels or toolbars.

"?" menu

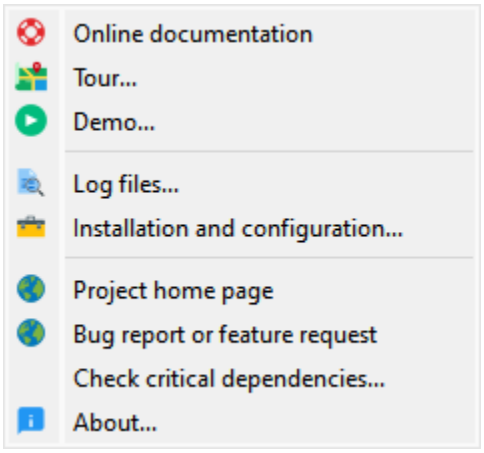
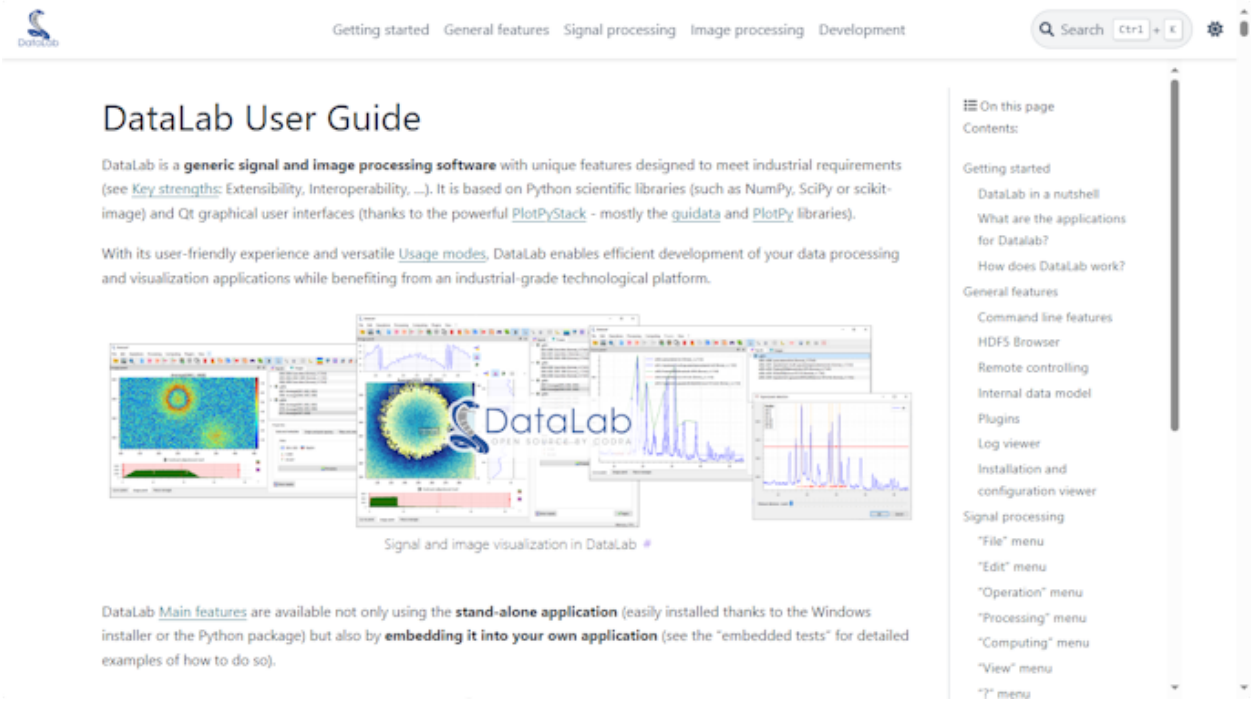


Fig. 20: Screenshot of the "?" menu.

The "?" menu allows you to access the online documentation, to show log files, to show information regarding your DataLab installation, and to show the "About DataLab" dialog box.

Online or Local documentation

Open the online or local documentation:



Show log files

Open DataLab log viewer

See also:

See *Log viewer* for more details on log viewer.

About DataLab installation

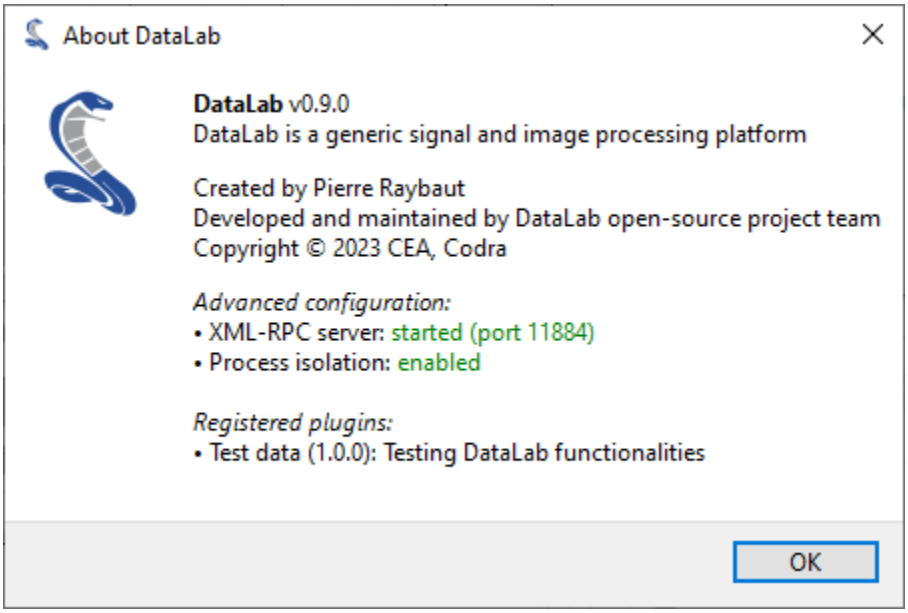
Show information regarding your DataLab installation (this is typically needed for submitting a bug report).

See also:

See *Installation and configuration viewer* for more details on this dialog box.

About

Open the "About DataLab" dialog box:



2.2.2 Signal Text File Import

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.

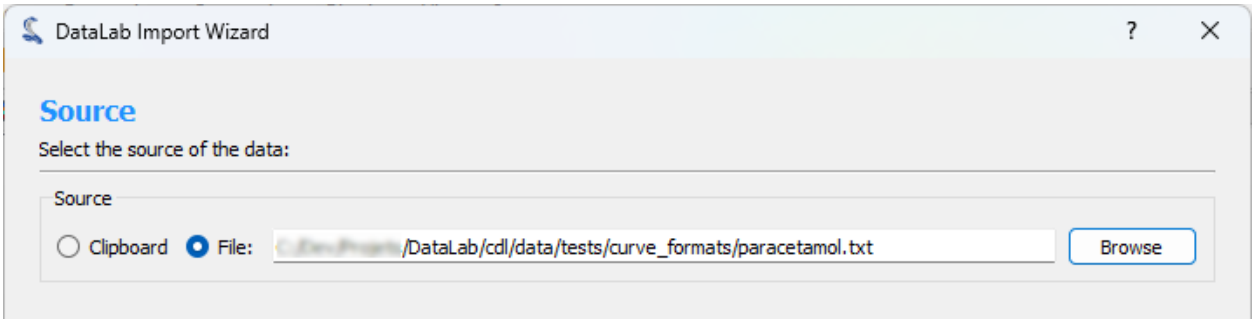


Fig. 21: Step 1: Select the source

Step 2: Preview and configure the import

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

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

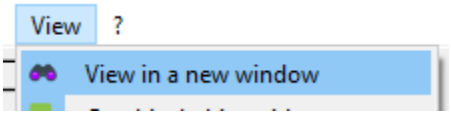
When you are done configuring the import, click the *Apply* button to see the result.

Step 3: Show graphical representation

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

2.2.3 Annotations (Signals)

DataLab provides an annotation feature for signals (as well as for images).



How to use the feature:

- Create or open a signal in DataLab workspace
- Double-click on the signal or select “View in a new window” in “View” menu

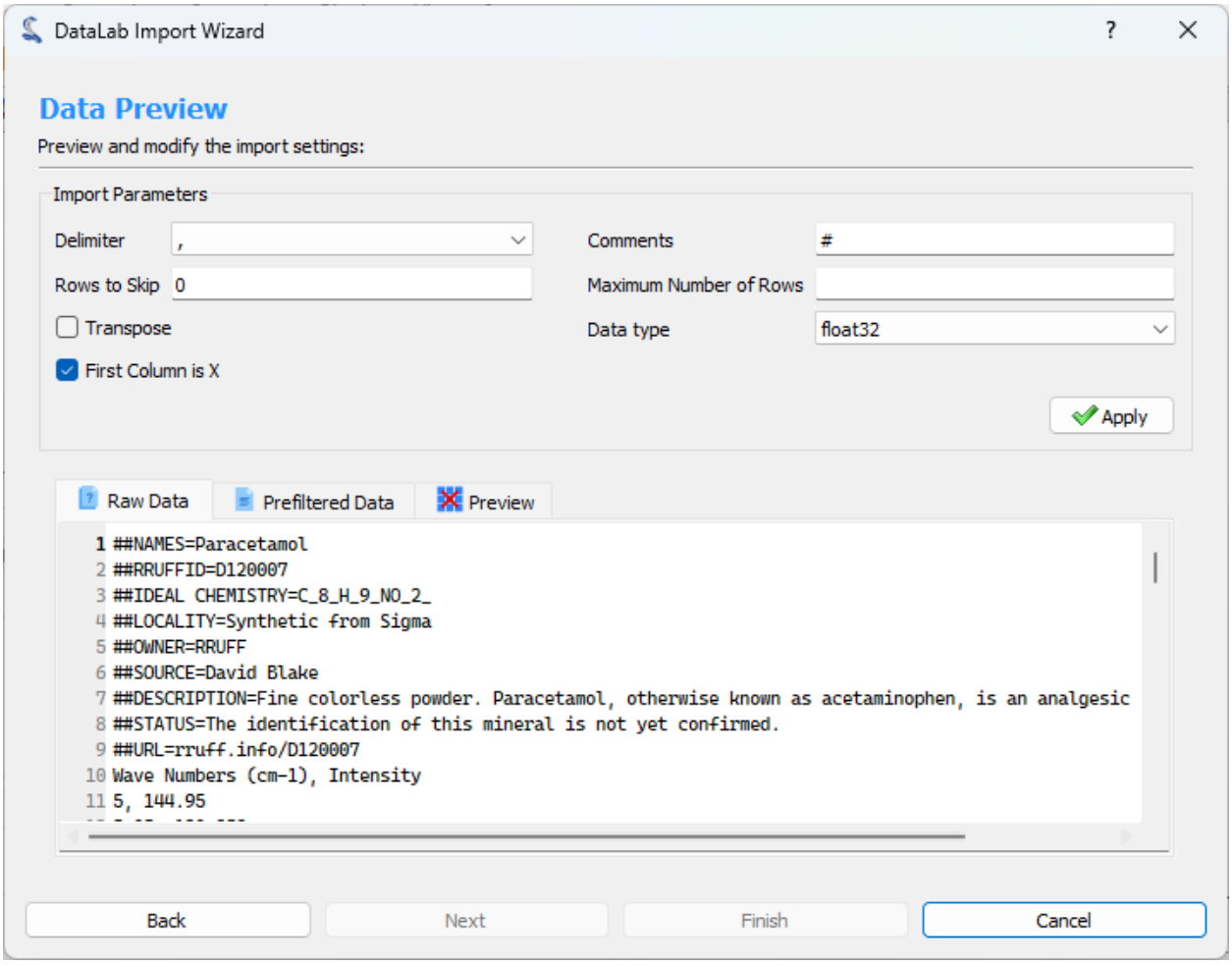


Fig. 22: Step 2: Configure the import

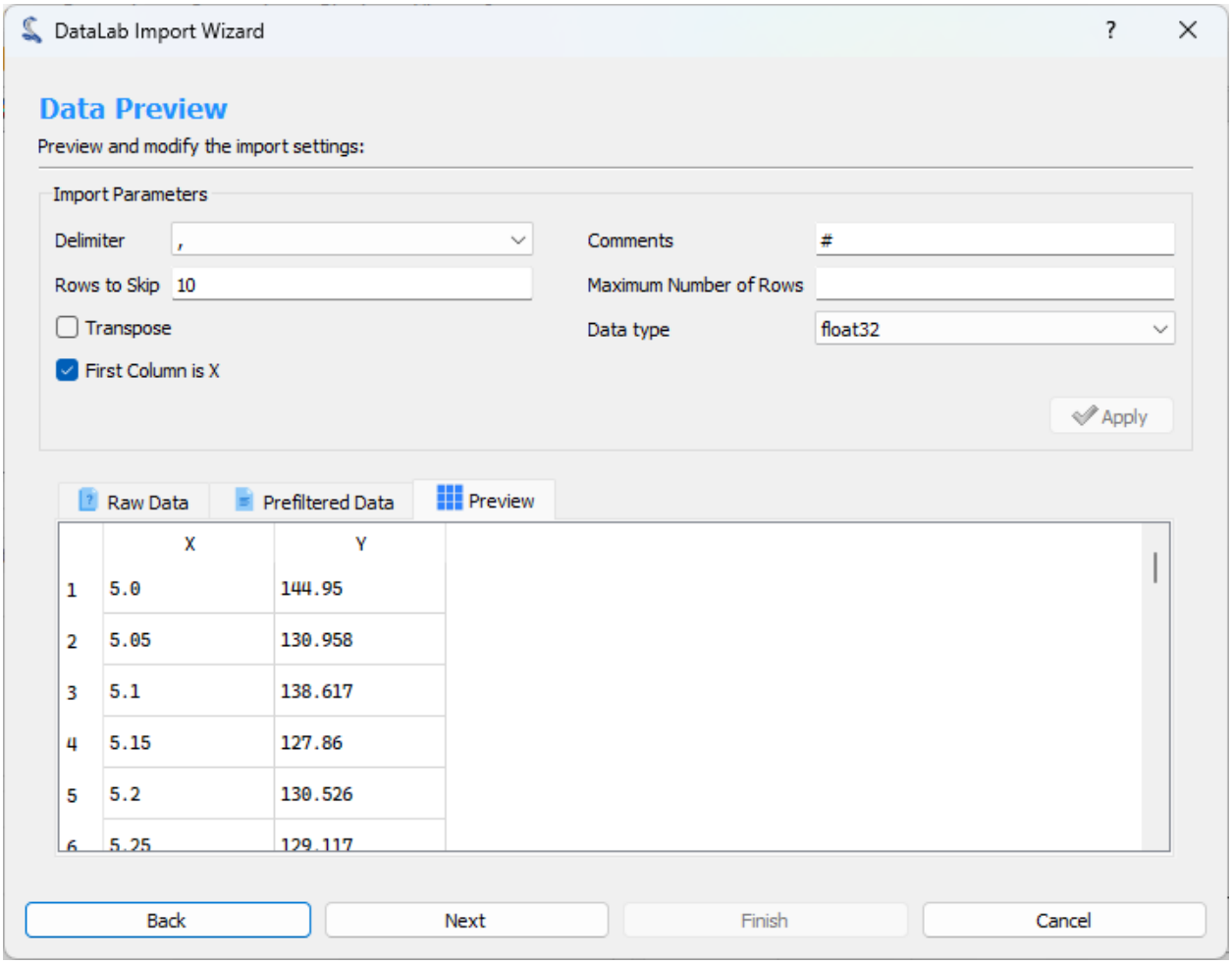


Fig. 23: Step 2: Preview the result

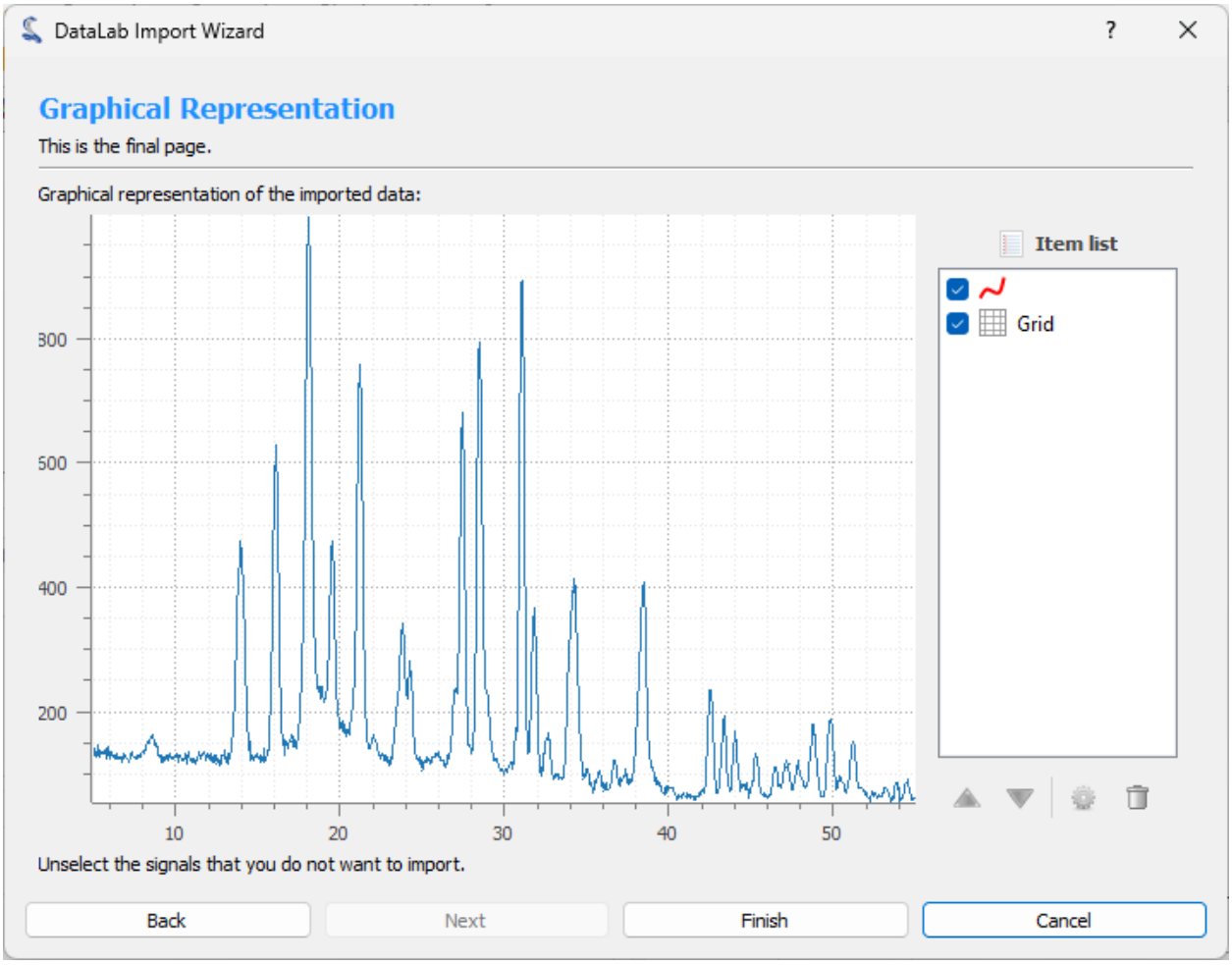


Fig. 24: Step 3: Show graphical representation

- Add annotations (labels, cursors, rectangles and segments)
- Eventually customize the annotations (right-click, “Parameters”)
- Validate your changes by clicking on “OK” button
- That’s it: your annotations are now attached to the signal and will be saved with your DataLab workspace

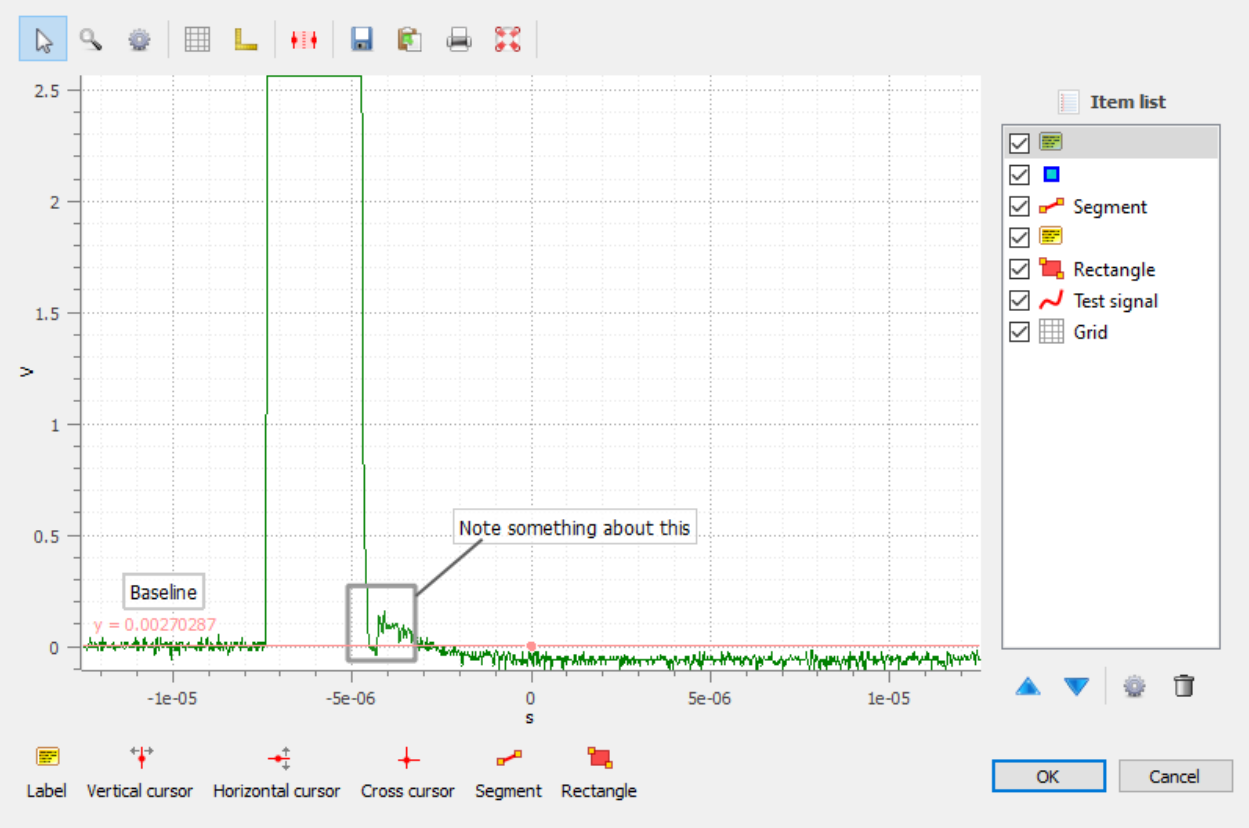


Fig. 25: Annotations may be added in the separate view.

Once the annotations have been added in the separate view (see above), they are part of the object (signal) metadata (see below).

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

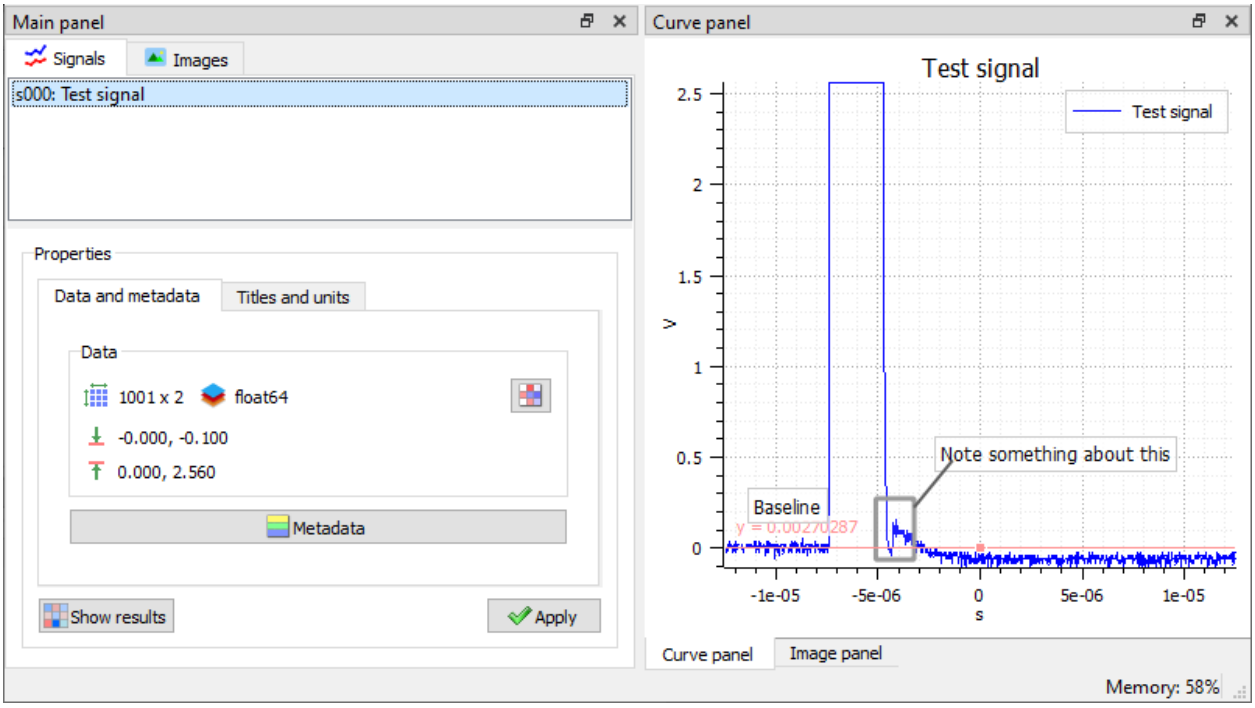


Fig. 26: Annotations are now part of the signal metadata.

2.3 Image processing

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

2.3.1 Menus

This section describes the image related features of DataLab, by presenting the different menus and their associated functions.

“File” menu

The “File” menu allows you to create, open, save and close images. It also allows you to import and export data from/to HDF5 files, and to edit the settings of the current session.

New image

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

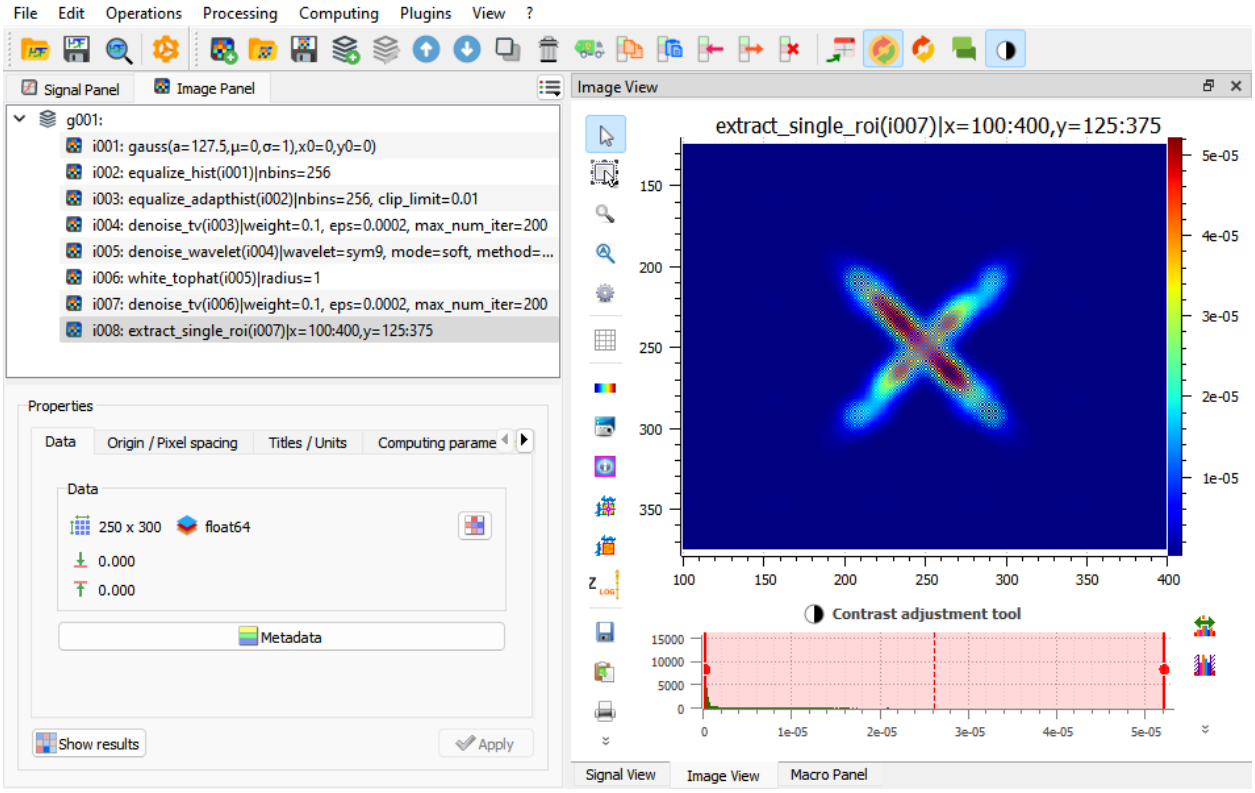


Fig. 27: DataLab main window: Image processing view

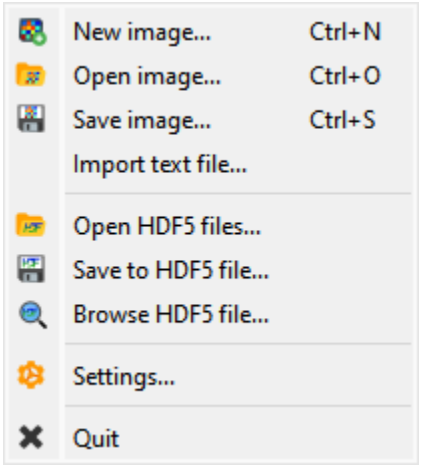


Fig. 28: Screenshot of the “File” menu.

Model	Equation
Zeros	$z[i] = 0$
Empty	Data is directly taken from memory as it is
Random	$z[i] \in [0, z_{max})$ where z_{max} is the datatype maximum value
2D Gaussian	$z = A.exp(-\frac{(\sqrt{(x-x0)^2+(y-y0)^2}-\mu)^2}{2\sigma^2})$

Open image

Create a new image from the following supported filetypes:

File type	Extensions
PNG files	.png
TIFF files	.tif, .tiff
8-bit images	.jpg, .gif
NumPy arrays	.npy
Text files	.txt, .csv, .asc
Andor SIF files	.sif
SPIRICON files	.scor-data
FXD files	.fxd
Bitmap images	.bmp

Save image

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

Import text file

Import data from a text file.

See also:

See *Image Text File Import* page for more details on importing text files.

Open HDF5 file

Import data from a HDF5 file.

Save to HDF5 file

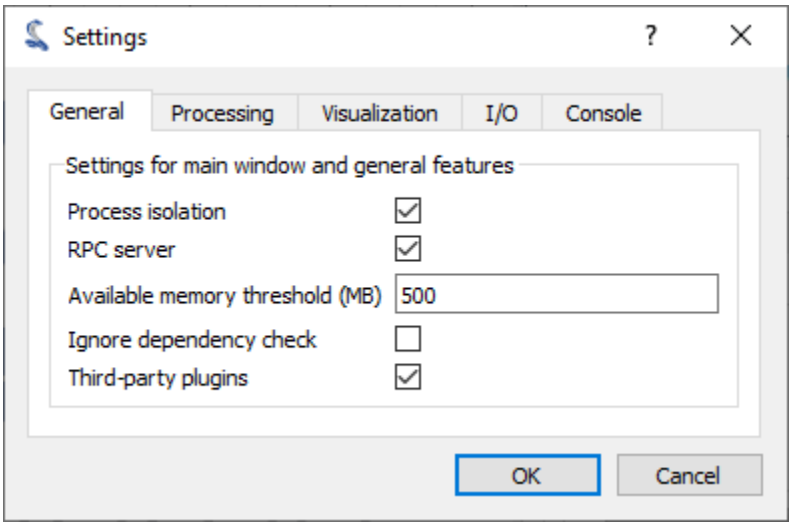
Export the whole DataLab session (all signals and images) into a HDF5 file.

Browse HDF5 file

Open the *HDF5 Browser* in a new window to browse and import data from HDF5 file.

Settings

Open the the “Settings” dialog box.



“Edit” menu

The “Edit” menu allows you to edit the current image or group of images, by adding, removing, renaming, moving up or down, or duplicating images. It also manipulates metadata, or handles image titles.

New group

Create a new group of images. Images may be moved from one group to another by drag and drop.

Rename group

Rename currently selected group.

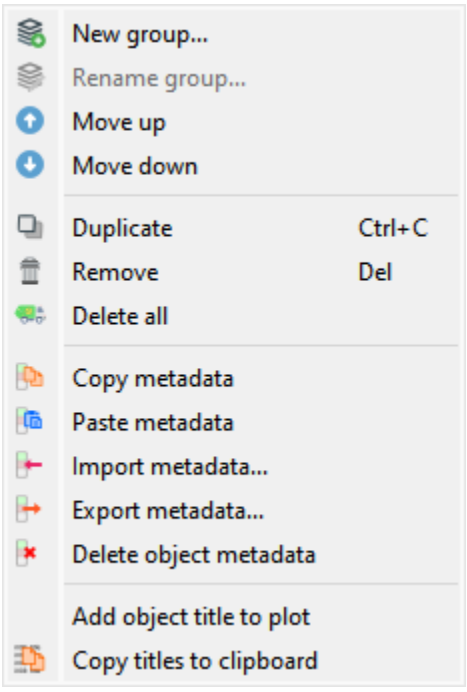


Fig. 29: Screenshot of the "Edit" menu.

Move up

Move current selection up in the list (groups or images may be selected). If multiple objects are selected, they are moved together. If a selected image is already at the top of its group, it is moved to the bottom of the previous group.

Move down

Move current selection down in the list (groups or images may be selected). If multiple objects are selected, they are moved together. If a selected image is already at the bottom of its group, it is moved to the top of the next group.

Duplicate

Create a new image which is identical to the currently selected object.

Remove

Remove currently selected image.

Delete all

Delete all images.

Copy metadata

Copy metadata from currently selected image into clipboard.

Paste metadata

Paste metadata from clipboard into selected image.

Import metadata into image

Import metadata from a JSON text file.

Export metadata from image

Export metadata to a JSON text file.

Delete object metadata

Delete metadata from currently selected image. Metadata contains additionnal information such as Region of Interest or results of computations

Add object title to plot

Add currently selected image title to the associated plot.

Copy titles to clipboard

Copy all image titles to clipboard as a multiline text. This text may be used for reproducing a processing chain, for example.

"Operation" menu

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

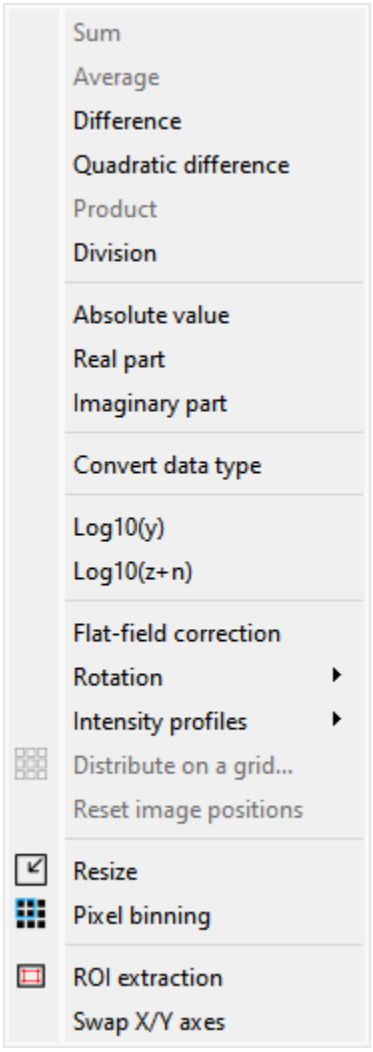


Fig. 30: Screenshot of the “Operation” menu.

Sum

Create a new image which is the sum of all selected images:

$$z_M = \sum_{k=0}^{M-1} z_k$$

Average

Create a new image which is the average of all selected images:

$$z_M = \frac{1}{M} \sum_{k=0}^{M-1} z_k$$

Difference

Create a new image which is the difference of the **two** selected images:

$$z_2 = z_1 - z_0$$

Quadratic difference

Create a new image which is the quadratic difference of the **two** selected images:

$$z_2 = \frac{z_1 - z_0}{\sqrt{2}}$$

Product

Create a new image which is the product of all selected images:

$$z_M = \prod_{k=0}^{M-1} z_k$$

Division

Create a new image which is the division of the **two** selected images:

$$z_2 = \frac{z_1}{z_0}$$

Absolute value

Create a new image which is the absolute value of each selected image:

$$z_k = |z_{k-1}|$$

Real part

Create a new image which is the real part of each selected image:

$$z_k = \Re(z_{k-1})$$

Imaginary part

Create a new image which is the imaginary part of each selected image:

$$z_k = \Im(z_{k-1})$$

Convert data type

Create a new image which is the result of converting data type of each selected image.

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

Log10(z)

Create a new image which is the base 10 logarithm of each selected image:

$$z_k = \log_{10}(z_{k-1})$$

Log10(z+n)

Create a new image which is the Log10(z+n) of each selected image (avoid Log10(0) on image background):

$$z_k = \log_{10}(z_{k-1} + n)$$

Flat-field correction

Create a new image which is flat-field correction of the **two** selected images:

$$z_1 = \begin{cases} \frac{z_0}{z_f}.\overline{z_f} & \text{if } z_0 > z_{threshold} \\ z_0 & \text{otherwise} \end{cases},$$

where z_0 is the raw image, z_f is the flat field image, $z_{threshold}$ is an adjustable threshold and $\overline{z_f}$ is the flat field image average value:

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

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

Rotation

Create a new image which is the result of rotating (90°, 270° or arbitrary angle) or flipping (horizontally or vertically) data.

Intensity profiles

Line profile
Extract an horizontal or vertical profile from each selected image, and create new signals from these profiles.

Average profile
Extract an horizontal or vertical profile averaged over a rectangular area, from each selected image, and create new signals from these profiles.

Radial profile extraction
Extract a radial profile from each selected image, and create new signals from these profiles.

The following parameters are available:

Parameter	Description
Center	Center around which the radial profile is computed: centroid, image center, or user-defined
X	X coordinate of the center (if user-defined), in pixels
Y	Y coordinate of the center (if user-defined), in pixels

Distribute on a grid

Distribute selected images on a regular grid.

Reset image positions

Reset selected image positions to first image (x0, y0) coordinates.

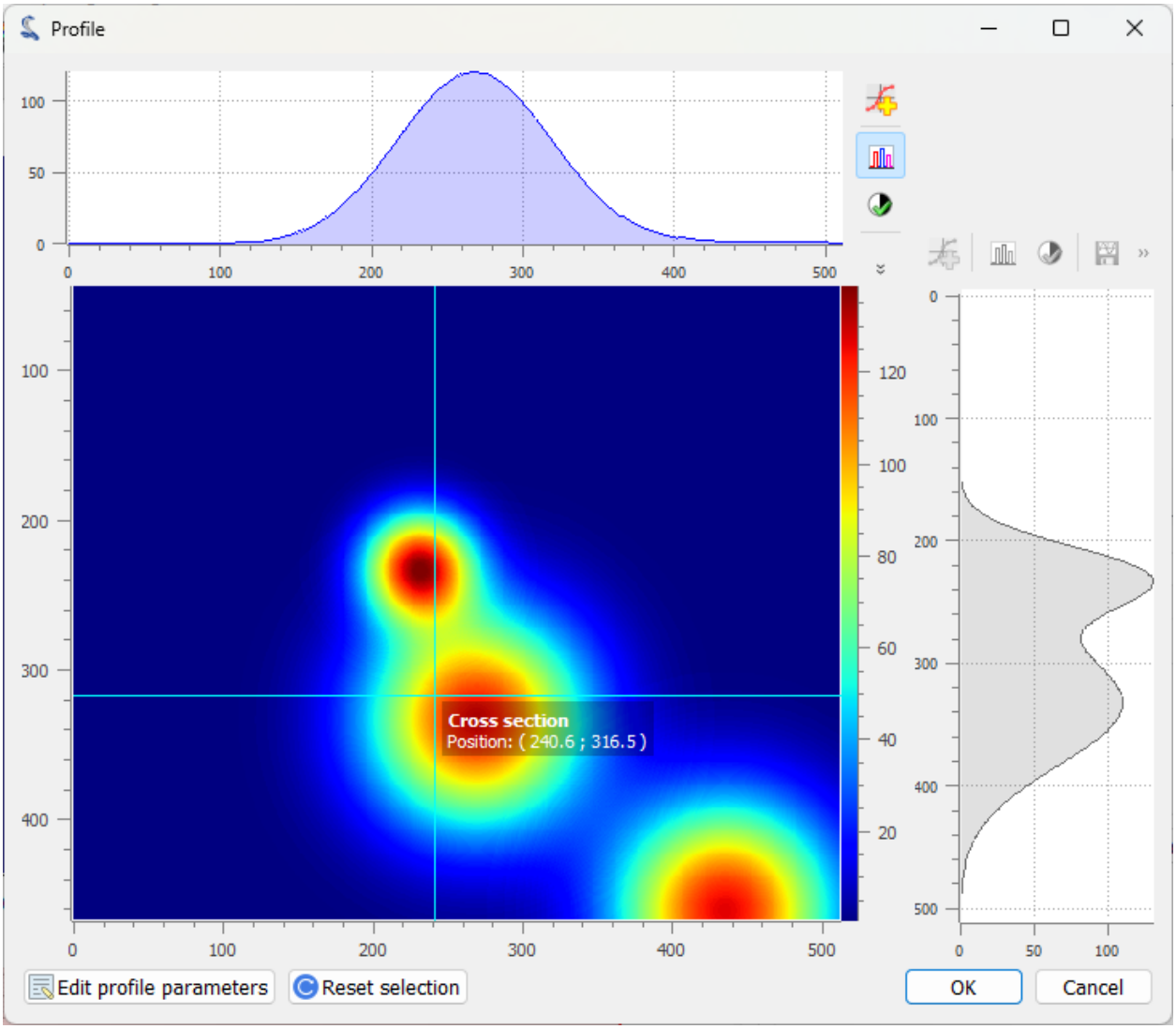


Fig. 31: Line profile dialog. Parameters may also be set manually (“Edit profile parameters” button).

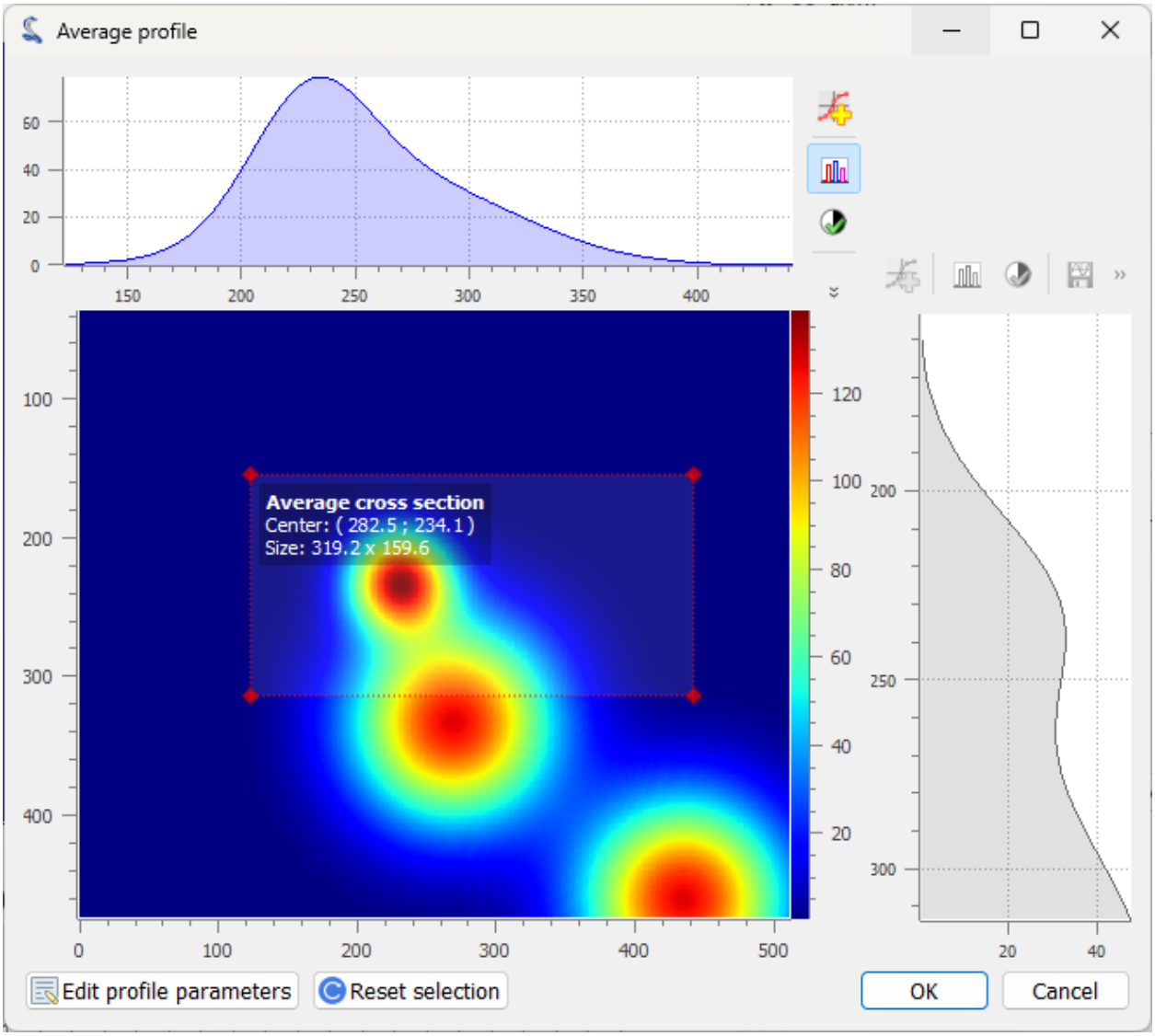


Fig. 32: Average profile dialog: the area is defined by a rectangle shape. Parameters may also be set manually ("Edit profile parameters" button).

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.

ROI extraction

Create a new image from a user-defined Region of Interest.

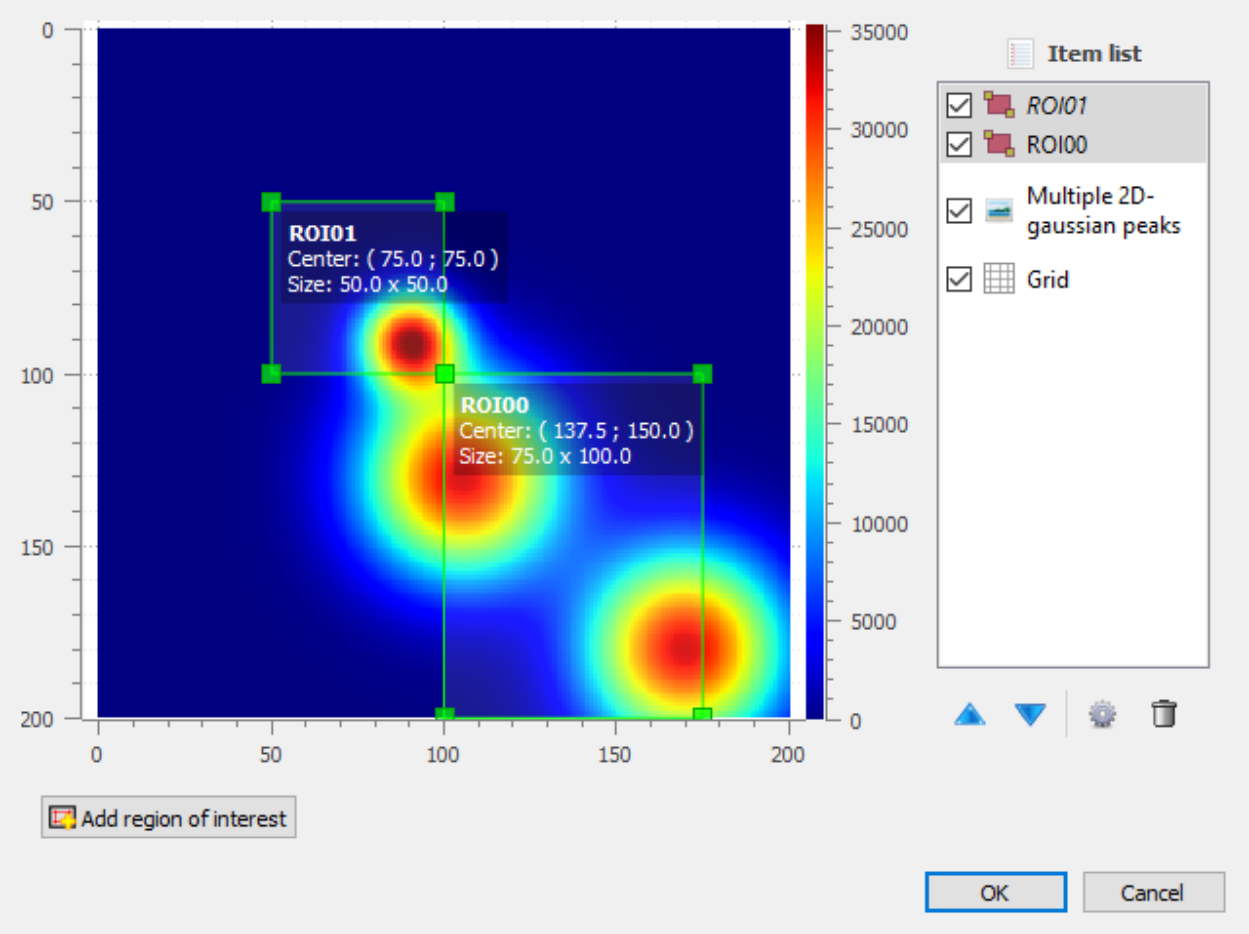


Fig. 33: ROI extraction dialog: the ROI is defined by moving the position and adjusting the size of a rectangle shape.

Swap X/Y axes

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

"Processing" menu

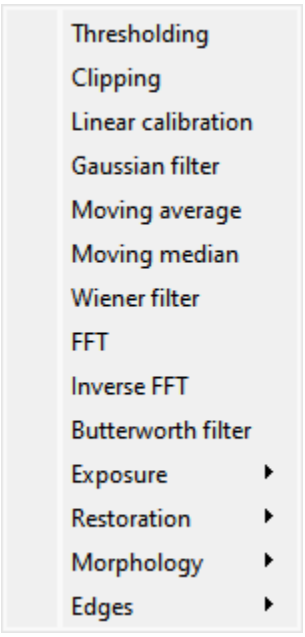


Fig. 34: Screenshot of the "Processing" menu.

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.

Linear calibration

Create a new image which is a linear calibration of each selected image with respect to Z axis:

Parameter	Linear calibration
Z-axis	$z_1 = a.z_0 + b$

Thresholding

Apply the thresholding to each selected image.

Clipping

Apply the clipping to each selected image.

Moving average

Compute moving average of each selected image (implementation based on scipy.ndimage.uniform_filter).

Moving median

Compute moving median of each selected image (implementation based on scipy.signal.medfilt).

Wiener filter

Compute Wiener filter of each selected image (implementation based on scipy.signal.wiener).

FFT

Create a new image which is the Fast Fourier Transform (FFT) of each selected image.

Inverse FFT

Create a new image which is the inverse FFT of each selected image.

Butterworth filter

Perform Butterworth filter on an image (implementation based on skimage.filters.butterworth)

Exposure

Gamma correction
: Apply gamma correction to each selected image (implementation based on skimage.exposure.adjust_gamma)

Logarithmic correction
: Apply logarithmic correction to each selected image (implementation based on skimage.exposure.adjust_log)

Sigmoid correction
: Apply sigmoid correction to each selected image (implementation based on skimage.exposure.adjust_sigmoid)

Histogram equalization
: Equalize image histogram levels (implementation based on skimage.exposure.equalize_hist)

Adaptive histogram equalization
: Equalize image histogram levels using Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm (implementation based on skimage.exposure.equalize_adapthist)

Intensity rescaling
: Stretch or shrink image intensity levels (implementation based on skimage.exposure.rescale_intensity)

Restoration

Total variation denoising
Denoise image using Total Variation algorithm (implementation based on skimage.restoration.denoise_tv_chambolle)

Bilateral filter denoising
Denoise image using bilateral filter (implementation based on skimage.restoration.denoise_bilateral)

Wavelet denoising
Perform wavelet denoising on image (implementation based on skimage.restoration.denoise_wavelet)

White Top-Hat denoising
Denoise image by subtracting its white top hat transform (using a disk footprint)

All denoising methods
Perform all denoising methods on image. Combined with the "distribute on a grid" option, this allows to compare the different denoising methods on the same image.

Morphology

White Top-Hat (disk)
Perform white top hat transform of an image, using a disk footprint (implementation based on skimage.morphology.white_tophat)

Black Top-Hat (disk)
Perform black top hat transform of an image, using a disk footprint (implementation based on skimage.morphology.black_tophat)

Erosion (disk)
Perform morphological erosion on an image, using a disk footprint (implementation based on skimage.morphology.erosion)

Dilation (disk)
Perform morphological dilation on an image, using a disk footprint (implementation based on skimage.morphology.dilation)

Opening (disk)
Perform morphological opening on an image, using a disk footprint (implementation based on skimage.morphology.opening)

Closing (disk)
Perform morphological closing on an image, using a disk footprint (implementation based on skimage.morphology.closing)

All morphological operations
Perform all morphological operations on an image, using a disk footprint. Combined with the "distribute on a grid" option, this allows to compare the different morphological operations on the same image.

Edges

Roberts filter
Perform edge filtering on an image, using the Roberts algorithm (implementation based on skimage.filters.roberts)

Prewitt filter
Perform edge filtering on an image, using the Prewitt algorithm (implementation based on skimage.filters.prewitt)

Prewitt filter (horizontal)
Find the horizontal edges of an image, using the Prewitt algorithm (implementation based on skimage.filters.prewitt_h)

Prewitt filter (vertical)
Find the vertical edges of an image, using the Prewitt algorithm (implementation based on skimage.filters.prewitt_v)

Sobel filter
Perform edge filtering on an image, using the Sobel algorithm (implementation based on skimage.filters.sobel)

Sobel filter (horizontal)
Find the horizontal edges of an image, using the Sobel algorithm (implementation based on skimage.filters.sobel_h)

Sobel filter (vertical)
Find the vertical edges of an image, using the Sobel algorithm (implementation based on skimage.filters.sobel_v)

Scharr filter
Perform edge filtering on an image, using the Scharr algorithm (implementation based on skimage.filters.scharr)

Scharr filter (horizontal)
Find the horizontal edges of an image, using the Scharr algorithm (implementation based on skimage.filters.scharr_h)

Scharr filter (vertical)
Find the vertical edges of an image, using the Scharr algorithm (implementation based on skimage.filters.scharr_v)

Farid filter
Perform edge filtering on an image, using the Farid algorithm (implementation based on skimage.filters.farid)

Farid filter (horizontal)
Find the horizontal edges of an image, using the Farid algorithm (implementation based on skimage.filters.farid_h)

Farid filter (vertical)
Find the vertical edges of an image, using the Farid algorithm (implementation based on skimage.filters.farid_v)

Laplace filter
Perform edge filtering on an image, using the Laplace algorithm (implementation based on skimage.filters.laplace)

All edges filters
Perform all edge filtering algorithms (see above) on an image. Combined with the "distribute on a grid" option, this allows to compare the different edge filters on the same image.

Canny filter
Perform edge filtering on an image, using the Canny algorithm (implementation based on skimage.feature.canny)

"Computing" menu

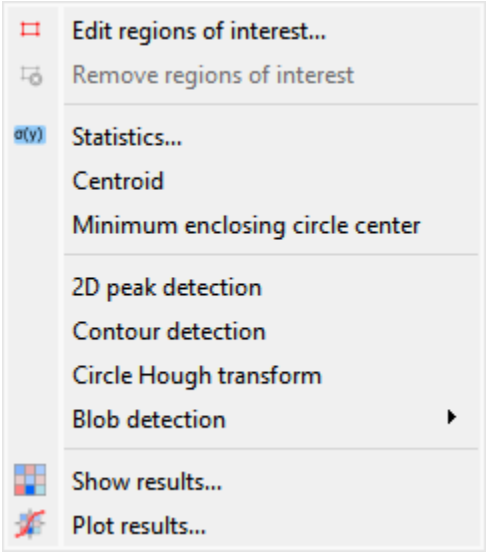


Fig. 35: Screenshot of the "Computing" menu.

The "Computing" 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, a "computing" 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.

Edit regions of interest

Open a dialog box to setup multiple Region Of Interests (ROI). ROI are stored as metadata, and thus attached to image.

ROI definition dialog is exactly the same as ROI extraction (see above).

Remove regions of interest

Remove all defined ROI for selected object(s).

Statistics

Compute statistics on selected image and show a summary table.

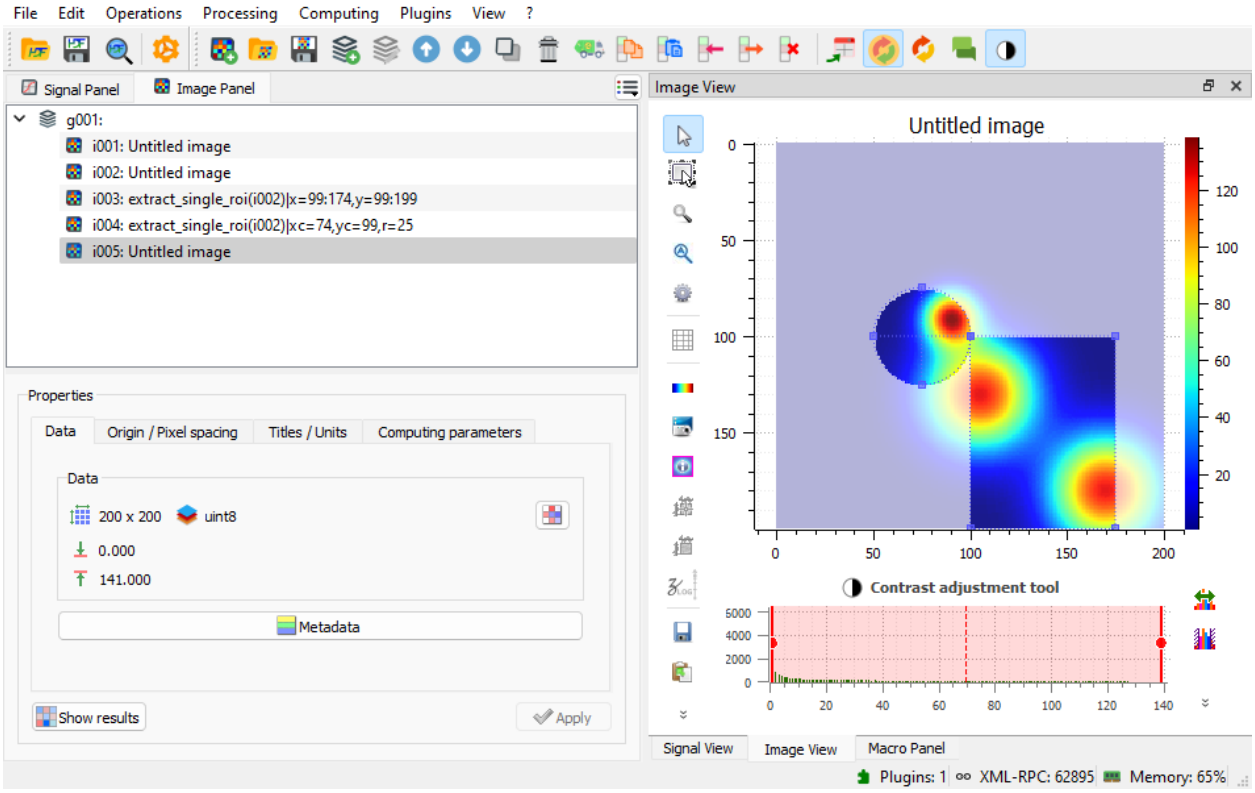


Fig. 36: An image with ROI.

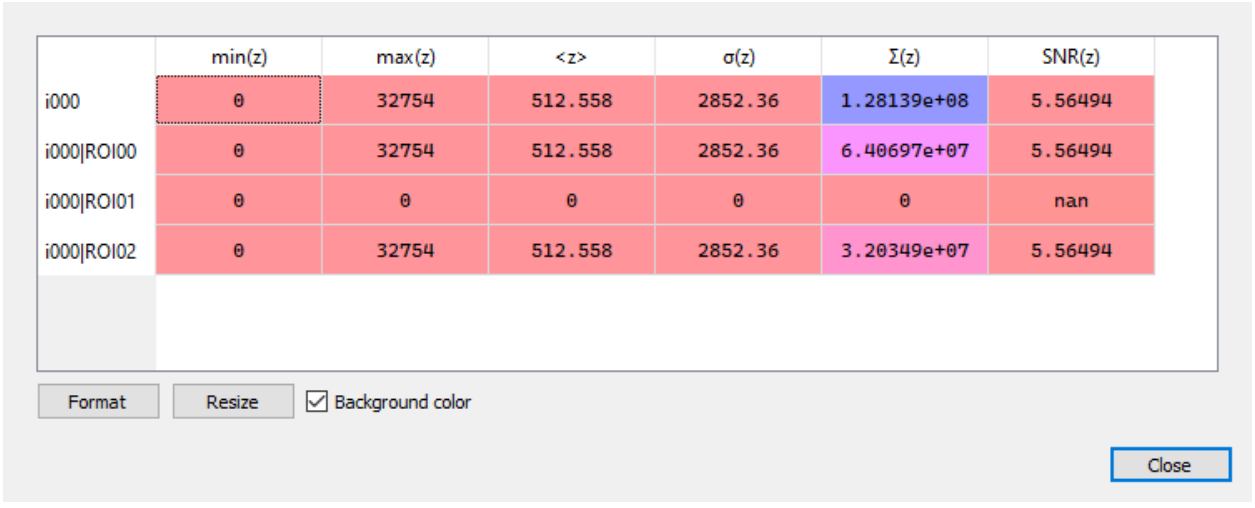


Fig. 37: Example of statistical summary table: each row is associated to an ROI (the first row gives the statistics for the whole data).

Centroid

Compute image centroid using a Fourier transform method (as discussed by Weisshaar et al.). This method is quite insensitive to background noise.

Minimum enclosing circle center

Compute the circle contour enclosing image values above a threshold level defined as the half-maximum value.

2D peak detection

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

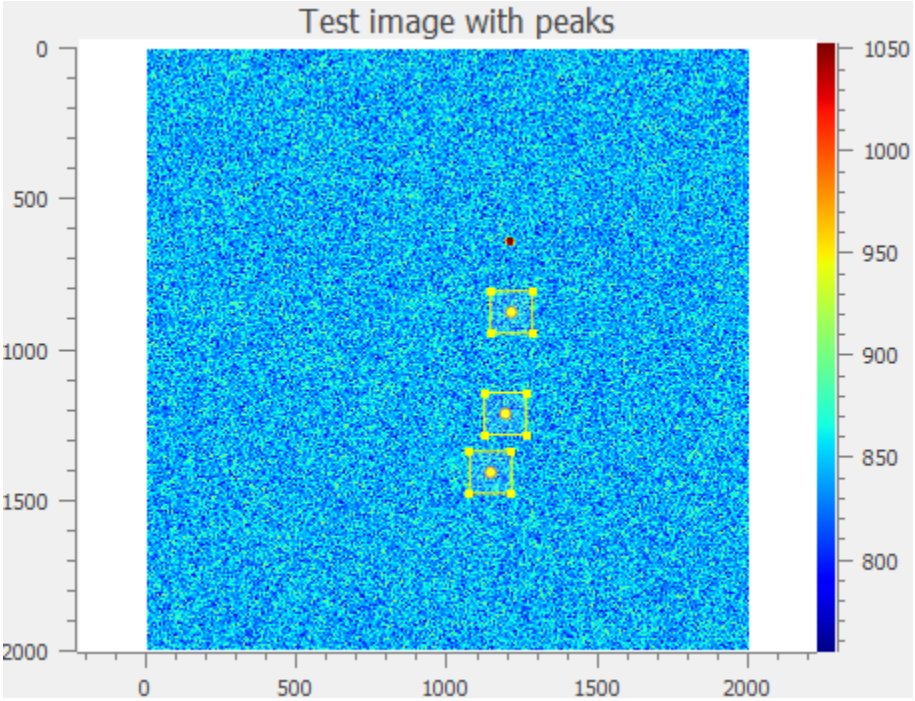


Fig. 38: Example of 2D peak detection.

See also:

See *2D Peak Detection* for more details on algorithm and associated parameters.

Contour detection

Automatically extract contours and fit them using a circle or an ellipse, or directly represent them as a polygon.

See also:

See *Contour Detection* for more details on algorithm and associated parameters.

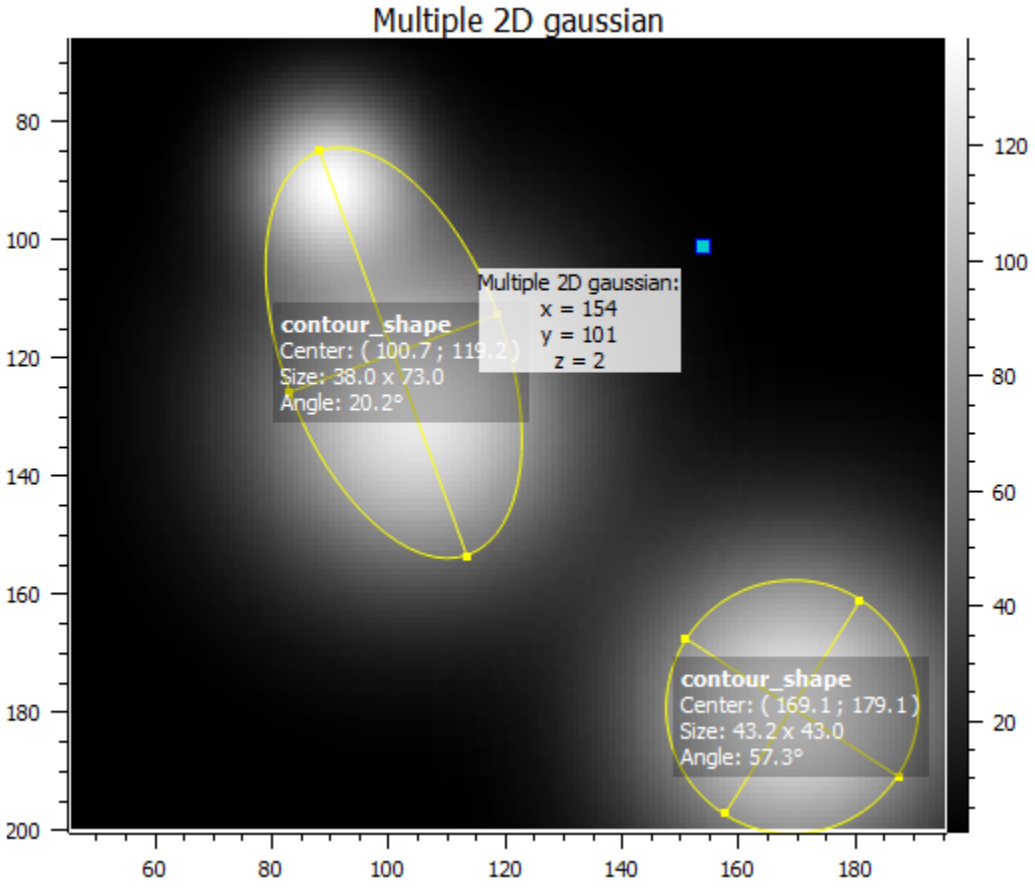


Fig. 39: Example of contour detection.

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.

Show results

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

Plot results

Plot the results of computations 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).

"View" menu

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 and the selected images.

In the separate window, you may visualize your data more comfortably (e.g., by maximizing the window) and you may also annotate the data.

See also:

See *Annotations (Images)* for more details on annotations.

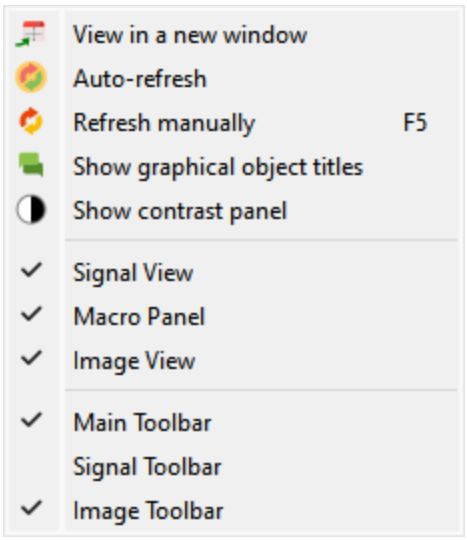


Fig. 40: Screenshot of the "View" menu.

Show graphical object titles

Show/hide titles of computing results or 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 clicking the "Refresh manually" button in the toolbar. 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.

Refresh manually

Refresh the visualization manually. This triggers a refresh of the plot view, even if the auto-refresh feature is disabled.

Show contrast panel

Show/hide contrast adjustment panel.

Other menu entries

Show/hide panels or toolbars.

"?" menu

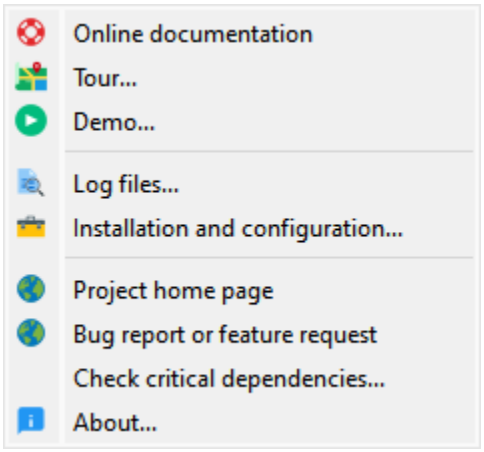
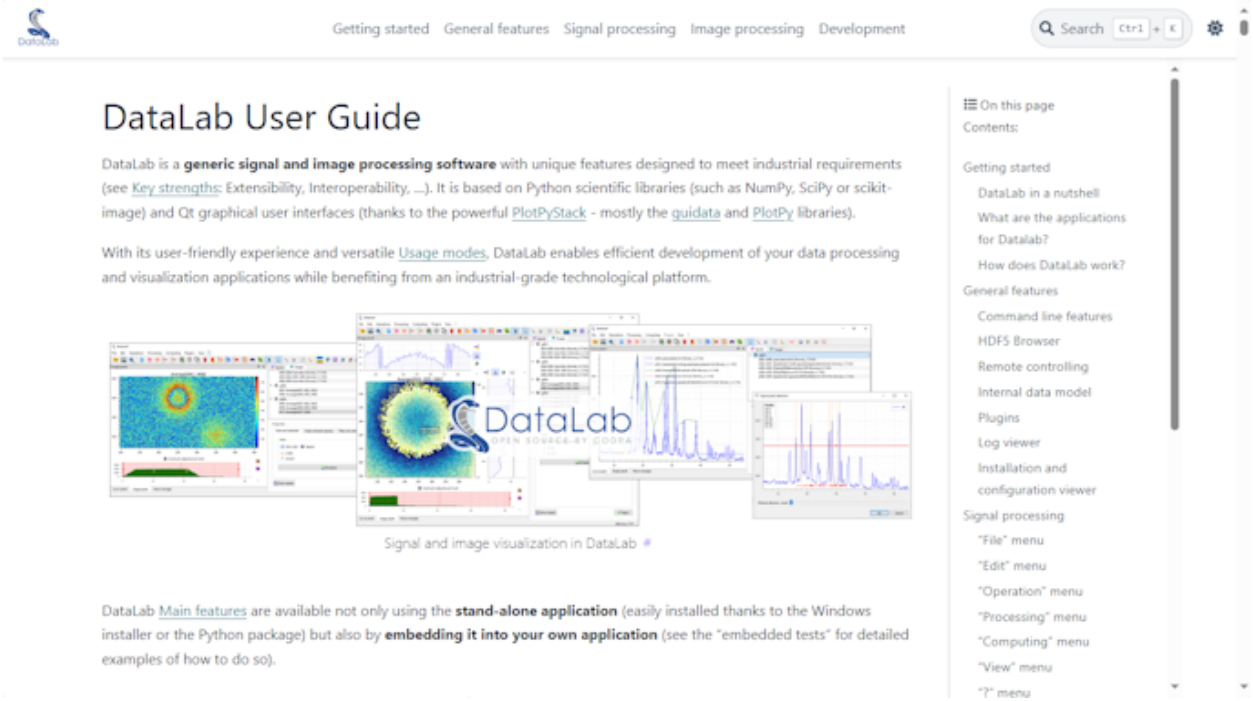


Fig. 41: Screenshot of the "?" menu.

The "?" menu allows you to access the online documentation, to show log files, to show information regarding your DataLab installation, and to show the "About DataLab" dialog box.

Online or Local documentation

Open the online or local documentation:



Show log files

Open DataLab log viewer

See also:

See *Log viewer* for more details on log viewer.

About DataLab installation

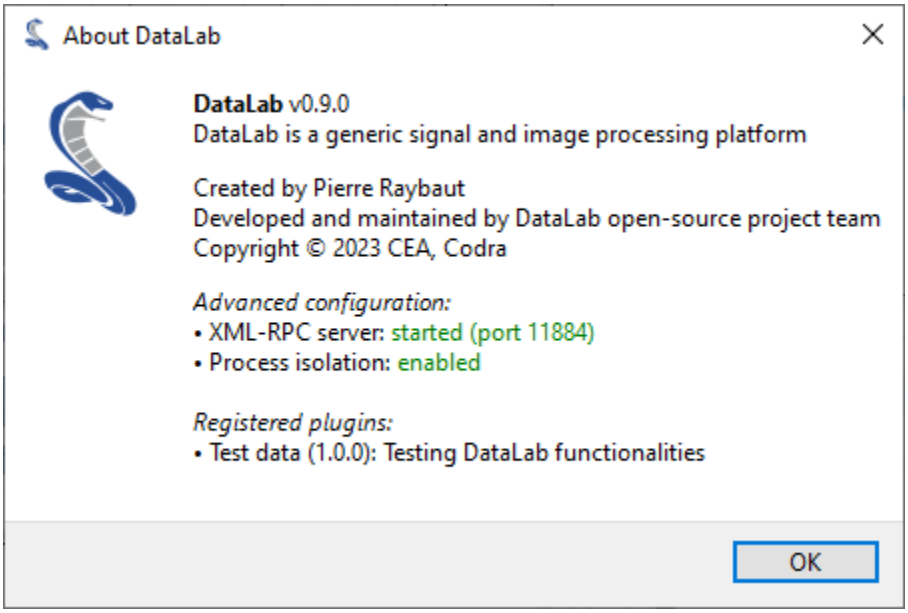
Show information regarding your DataLab installation (this is typically needed for submitting a bug report).

See also:

See *Installation and configuration viewer* for more details on this dialog box.

About

Open the "About DataLab" dialog box:



2.3.2 Image Text File Import

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.

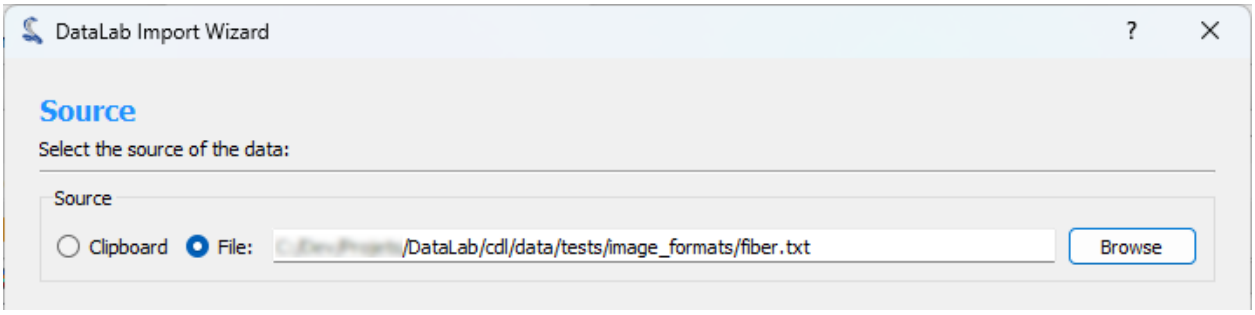


Fig. 42: Step 1: Select the source

Step 2: Preview and configure the import

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

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

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

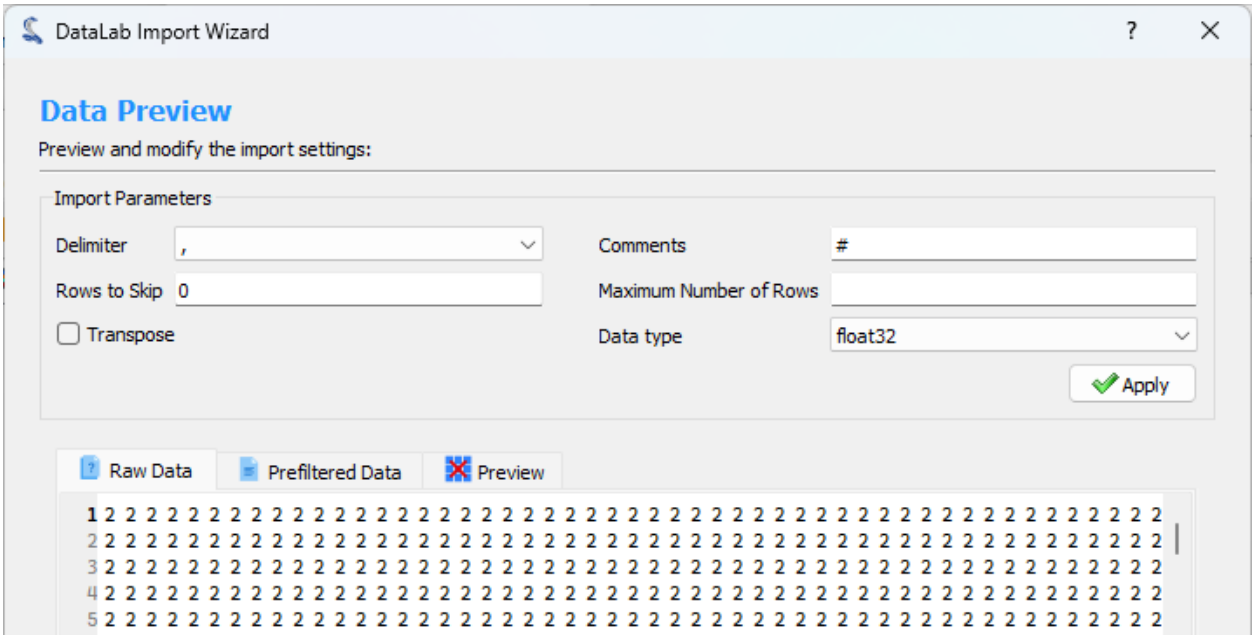


Fig. 43: Step 2: Configure the import

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.

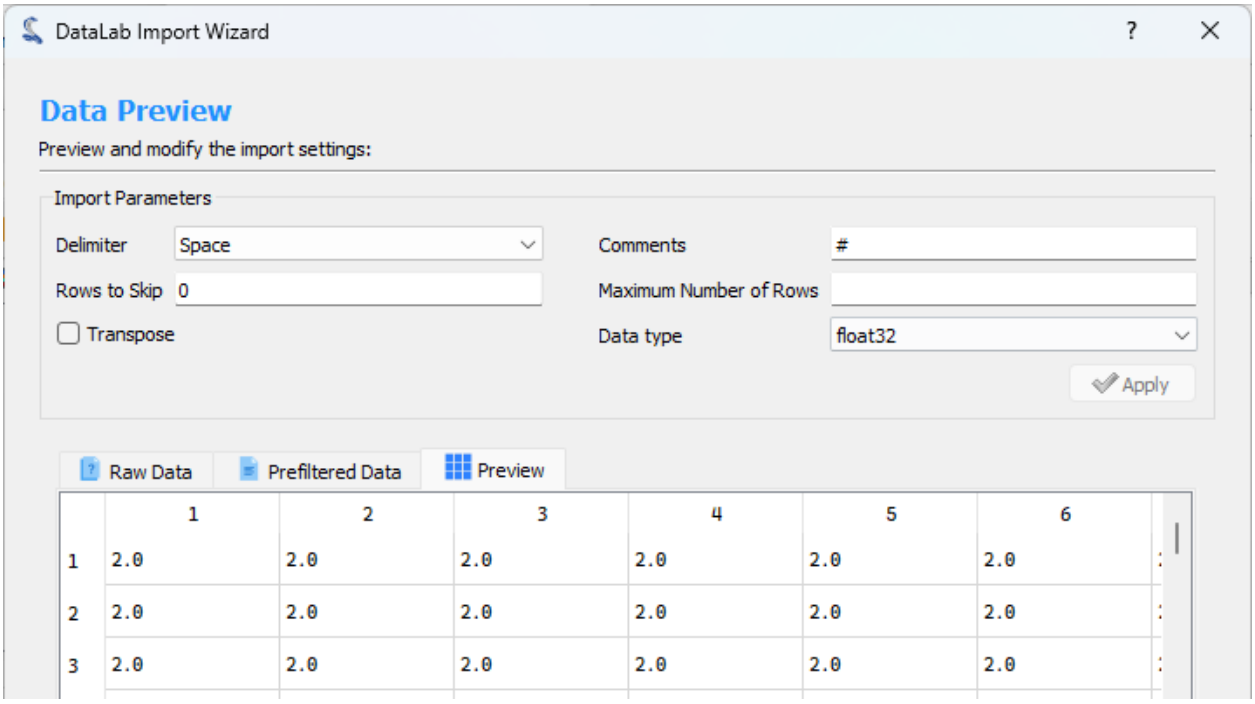


Fig. 44: Step 2: Preview the result

2.3.3 2D Peak Detection

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

How to use the feature:

- Create or open an image in DataLab workspace
- Select "2d peak detection" in "Computing" menu
- Enter parameters "Neighborhoods size" and "Relative threhold"
- Check "Create regions of interest" if you want a ROI defined for each detected peak (this may become useful when using another computation afterwards on each area around peaks, e.g. contour detection)

Results are shown in a table:

- Each row is associated to a detected peak
- First column shows the ROI index (0 if no ROI is defined on input image)
- Second and third columns show peak coordinates

The 2d peak detection algorithm works in the following way:

- First, the minimum and maximum filtered images are computed using a sliding window algorithm with a user-defined size (implementation based on scipy.ndimage.minimum_filter and scipy.ndimage.maximum_filter)
- 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

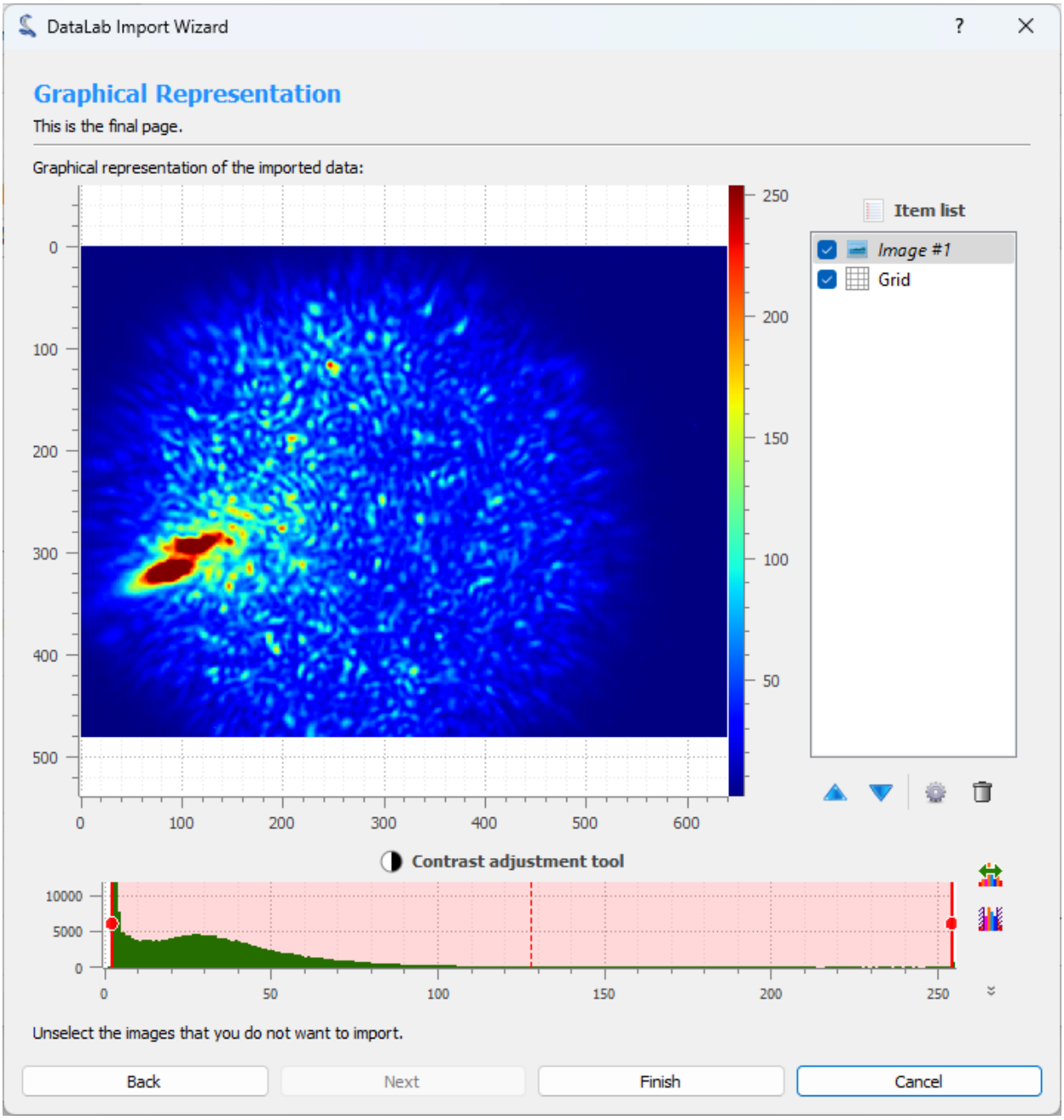


Fig. 45: Step 3: Show graphical representation

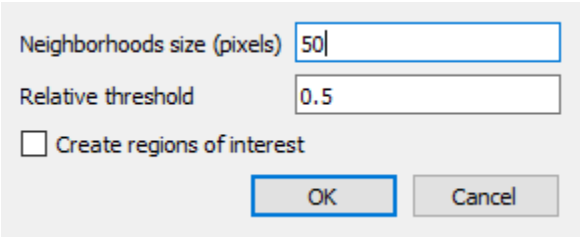


Fig. 46: 2D peak detection parameters.

Fig. 47: 2d peak detection results (see test "peak2d_app.py")

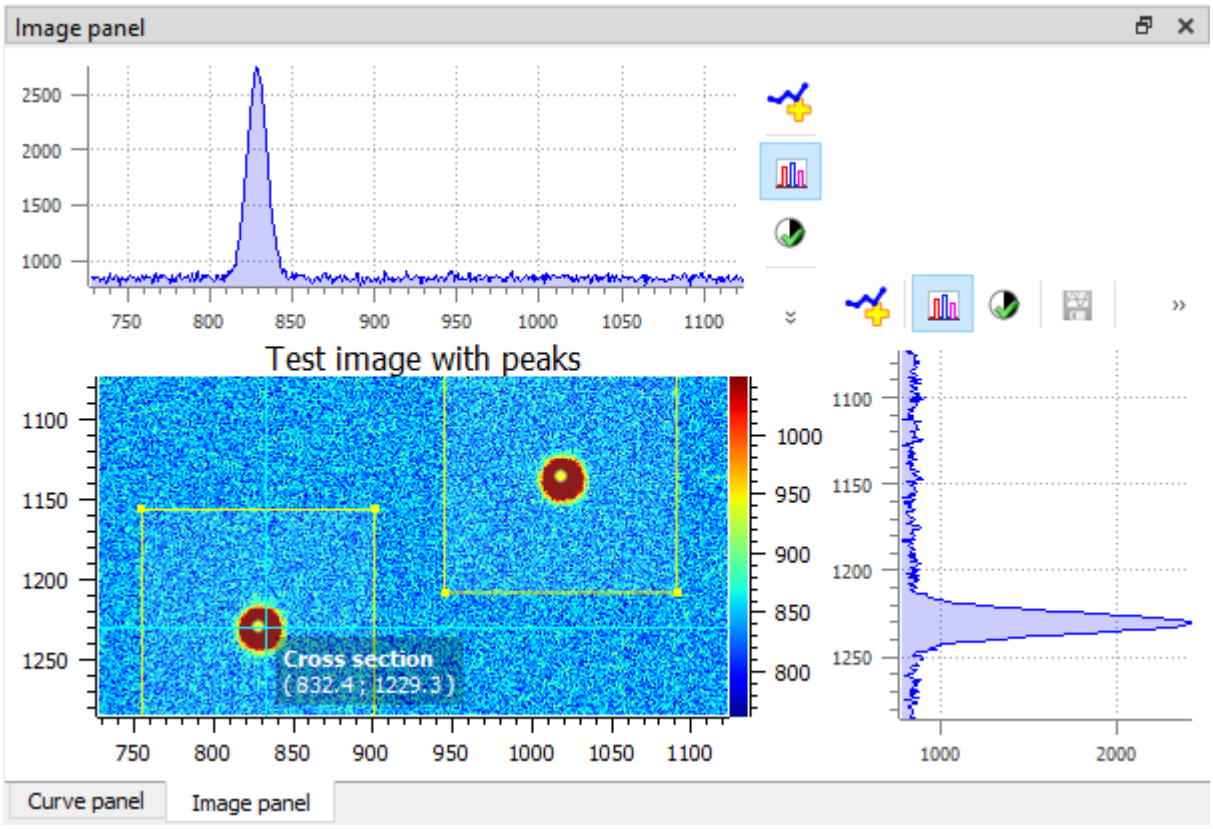


Fig. 48: Example of 2D peak detection.

- Duplicates are eventually removed

The 2d peak detection parameters are the following:

- "Neighborhoods size": size of the sliding window (see above)
- "Relative threshold": detection threshold

Feature is based on `get_2d_peaks_coords` function from `cdl.algorithms` module:

```
def get_2d_peaks_coords(
    data: np.ndarray, size: int | None = None, level: float = 0.5
) -> np.ndarray:
    """Detect peaks in image data, return coordinates.

    If neighborhoods size is None, default value is the highest value
    between 50 pixels and the 1/40th of the smallest image dimension.

    Detection threshold level is relative to difference
    between data maximum and minimum values.

    Args:
        data: Input data
        size: Neighborhood size (default: None)
        level: Relative level (default: 0.5)

    Returns:
        Coordinates of peaks
    """
    if size is None:
        size = max(min(data.shape) // 40, 50)
    data_max = spf.maximum_filter(data, size)
    data_min = spf.minimum_filter(data, size)
    data_diff = data_max - data_min
    diff = (data_max - data_min) > get_absolute_level(data_diff, level)
    maxima = data == data_max
    maxima[diff == 0] = 0
    labeled, _num_objects = spi.label(maxima)
    slices = spi.find_objects(labeled)
    coords = []
    for dy, dx in slices:
        x_center = int(0.5 * (dx.start + dx.stop - 1))
        y_center = int(0.5 * (dy.start + dy.stop - 1))
        coords.append((x_center, y_center))
    if len(coords) > 1:
        # Eventually removing duplicates
        dist = distance_matrix(coords)
        for index in reversed(np.unique(np.where((dist < size) & (dist >␣
↪0))[1])):
            coords.pop(index)
    return np.array(coords)
```

2.3.4 Contour Detection

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

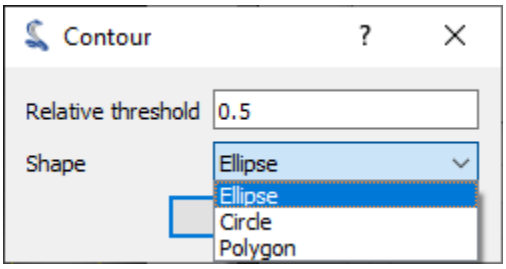


Fig. 49: Contour detection parameters.

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 "Computing" menu
- Enter parameter "Shape" ("Ellipse", "Circle" or "Polygon")

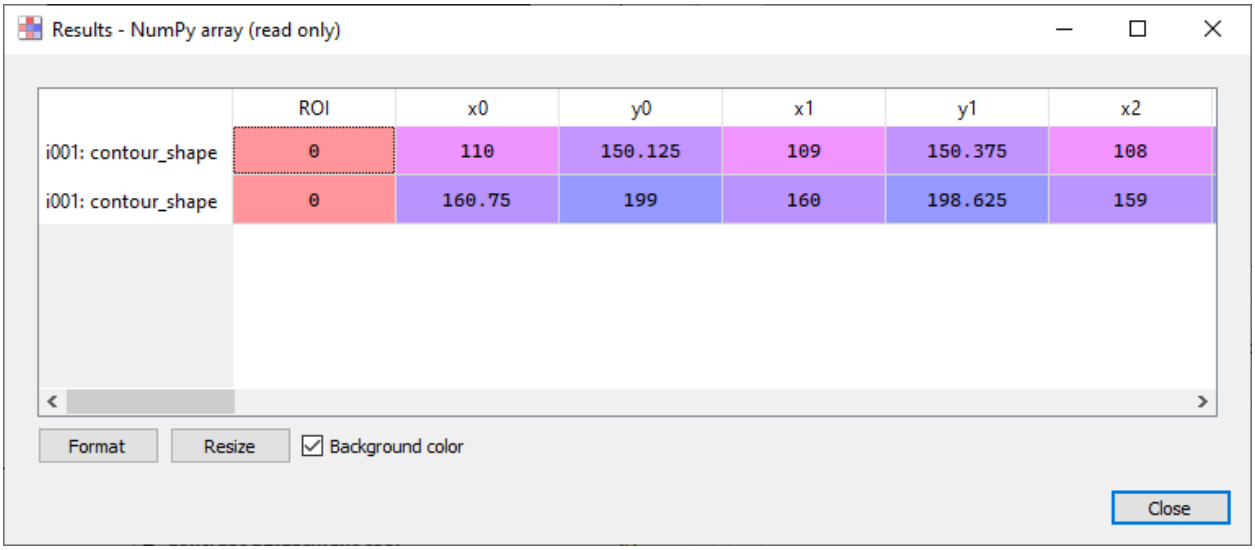


Fig. 50: Contour detection results (see test "contour_app.py")

Results are shown in a table:

- Each row is associated to a contour
- First column shows the ROI index (0 if no ROI is defined on input image)
- Other columns show contour coordinates: 4 columns for circles (coordinates of diameter), 8 columns for ellipses (coordinates of diameters)

The contour detection algorithm works in the following way:

- First, iso-valued contours are computed (implementation based on skimage.measure.find_contours.find_contours)

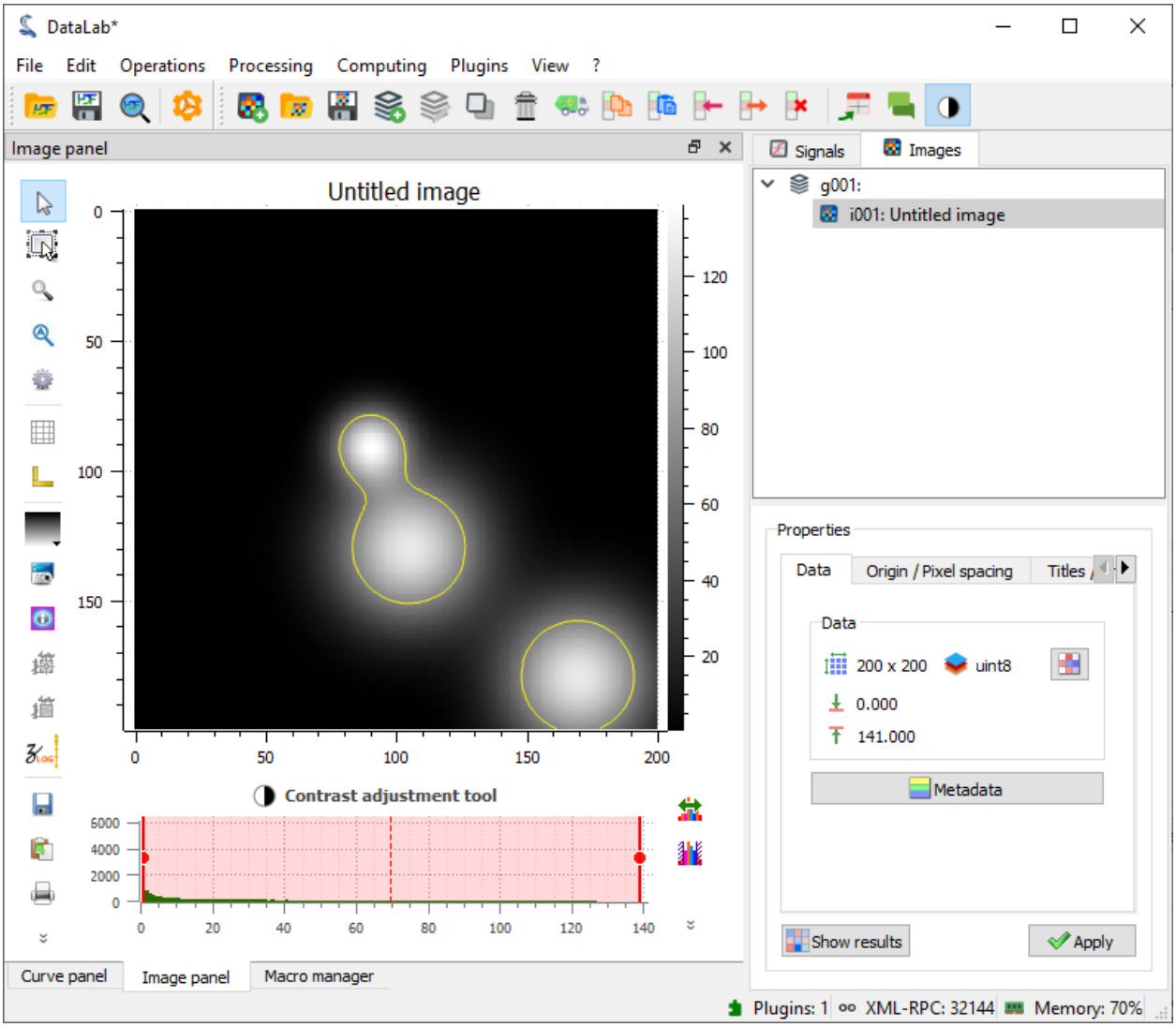


Fig. 51: Example of contour detection.

- Then, each contour is fitted to the closest ellipse (or circle)

Feature is based on `get_contour_shapes` function from `cdl.algorithms` module:

```python
def get_contour_shapes(
    data: np.ndarray | ma.MaskedArray, shape: str = "ellipse", level: float = 0.5
) -> np.ndarray:
    """Find iso-valued contours in a 2D array, above relative level (.5 means FWHM),
    then fit contours with shape ('ellipse' or 'circle')

    Args:
        data: Input data
        shape: Shape to fit. Valid values: 'circle', 'ellipse', 'polygon'.
         (default: 'ellipse')
        level: Relative level (default: 0.5)

    Returns:
        Coordinates of shapes
    """
    # pylint: disable=too-many-locals
    assert shape in ("circle", "ellipse", "polygon")
    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 == "circle":
            model = measure.CircleModel()
            if model.estimate(contour):
                yc, xc, r = model.params
                if r <= 1.0:
                    continue
                coords.append([xc, yc, r])
        elif shape == "ellipse":
            model = measure.EllipseModel()
            if model.estimate(contour):
                yc, xc, b, a, theta = model.params
                if a <= 1.0 or b <= 1.0:
                    continue
                coords.append([xc, yc, a, b, theta])
        elif shape == "polygon":
            # `contour` is a (N, 2) array (rows, cols): we need to convert it
            # to a list of x, y coordinates flattened in a single list
            coords.append(contour[:, ::-1].flatten())
        else:
            raise NotImplementedError(f"Invalid contour model {model}")
```

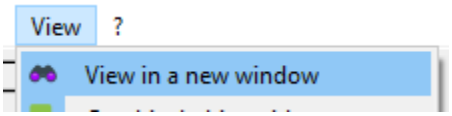
(continues on next page)

(continued from previous page)

```
    if shape == "polygon":
        # `coords` is a list of arrays of shape (N, 2) where N is the number of␣
↪points
        # that can vary from one array to another, so we need to padd with␣
↪NaNs each
        # array to get a regular array:
        max_len = max(coord.shape[0] for coord in coords)
        arr = np.full((len(coords), max_len), np.nan)
        for i_row, coord in enumerate(coords):
            arr[i_row, : coord.shape[0]] = coord
        return arr
    return np.array(coords)
```

2.3.5 Annotations (Images)

DataLab provides an annotation feature for images (as well as for signals).



How to use the feature:

- Create or open an image in DataLab workspace
- Double-click on the image or select “View in a new window” in “View” menu
- Add annotations (labels, rectangles, circles, etc.)
- Eventually customize the annotations (right-click, “Parameters”)
- Validate your changes by clicking on “OK” button
- That’s it: your annotations are now attached to the image and will be saved with your DataLab workspace

Once the annotations have been added in the separate view (see above), they are part of the object (image) metadata (see below).

Note: Annotations may be copied from an image to another by using the “copy/paste metadata” features.

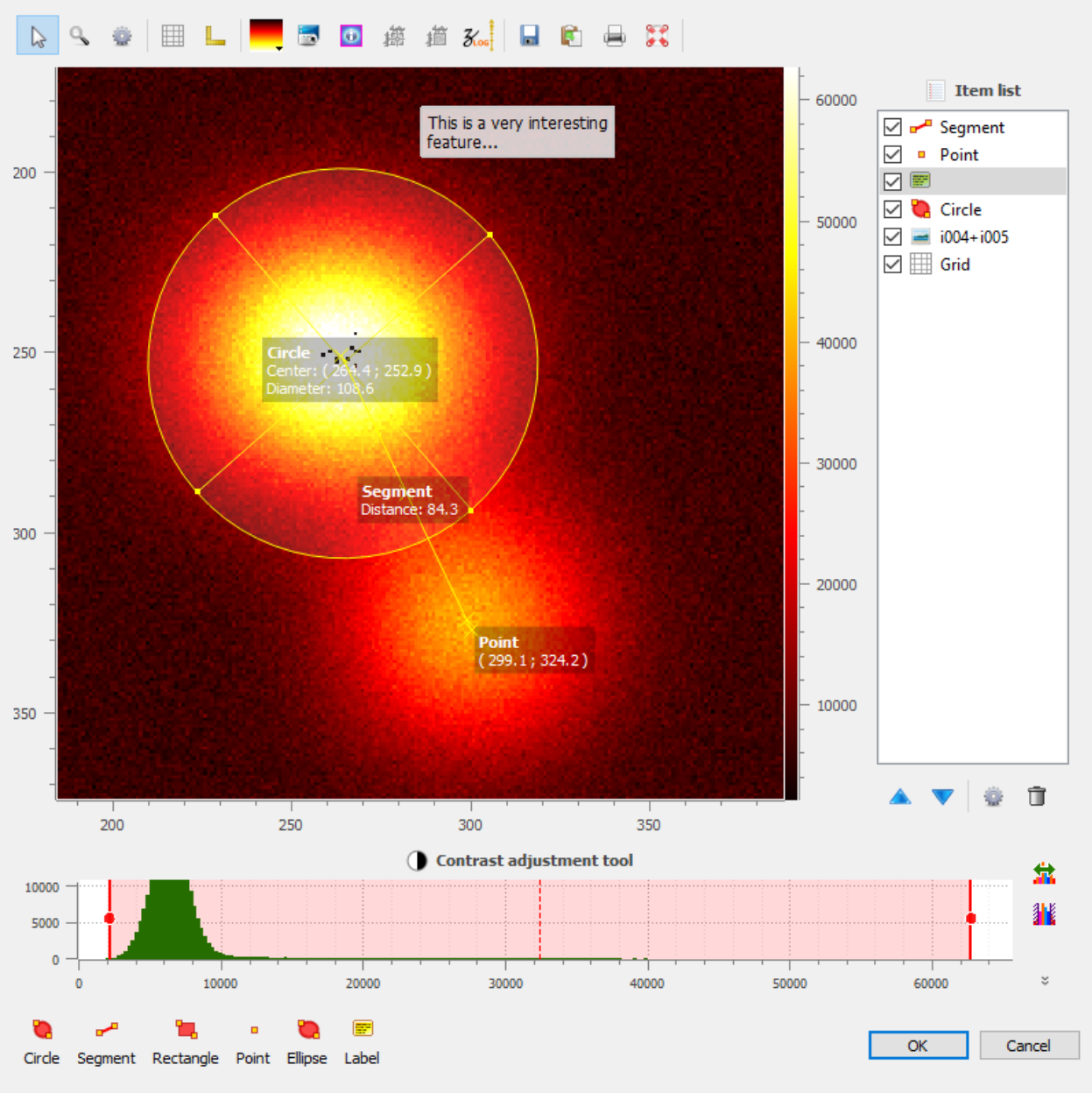


Fig. 52: Annotations may be added in the separate view.

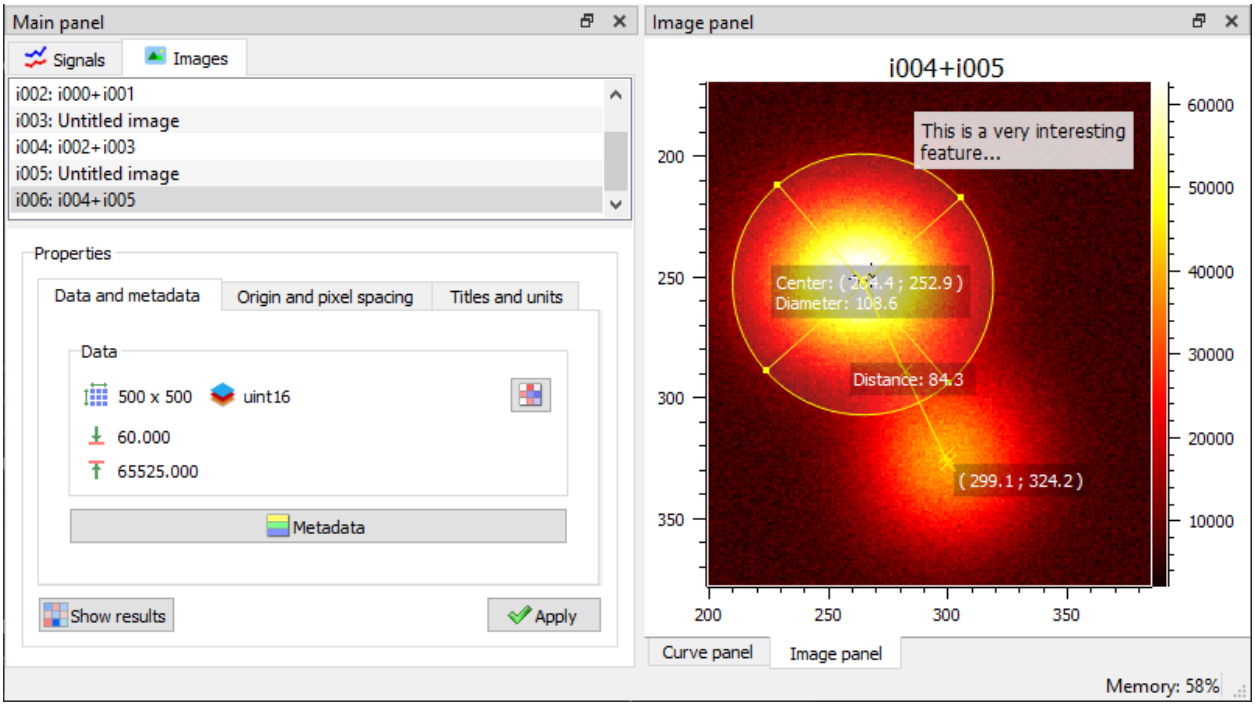


Fig. 53: Annotations are now part of the image metadata.

CHAPTER

THREE

API

The public Application Programming Interface (API) of DataLab offers a set of functions to access the DataLab features. Those functions are available in various submodules of the *cdl* package. The following table lists the available submodules and their purpose:

Submodule	Purpose
cdl.algorithms	Algorithms for data analysis, which operates on NumPy arrays
cdl.param	Convenience module to access the DataLab sets of parameters (instances of guidata.dataset.DataSet objects)
cdl.obj	Convenience module to access the DataLab objects (*cdl.obj.SignalObj* or *cdl.obj.ImageObj*) and related functions
cdl.core.computation	Computation functions, which operate on DataLab objects (*cdl.obj.SignalObj* or *cdl.obj.ImageObj*)
cdl.proxy	Proxy objects to access the DataLab interface from a Python script or a remote application

3.1 Algorithms (cdl.algorithms)

This package contains the algorithms used by the DataLab project. Those algorithms operate directly on NumPy arrays and are designed to be used in the DataLab pipeline, but can be used independently as well.

See also:

The *cdl.algorithms* package is the main entry point for the DataLab algorithms when manipulating NumPy arrays. See the *cdl.core.computation* package for algorithms that operate directly on DataLab objects (i.e. *cdl.obj.SignalObj* and *cdl.obj.ImageObj*).

The algorithms are organized in subpackages according to their purpose. The following subpackages are available:

- *cdl.algorithms.signal*: Signal processing algorithms
- *cdl.algorithms.image*: Image processing algorithms
- *cdl.algorithms.datatypes*: Data type conversion algorithms
- *cdl.algorithms.coordinates*: Coordinate conversion algorithms
- *cdl.algorithms.fit*: Fitting algorithms

3.1.1 Signal Processing Algorithms

cdl.algorithms.signal.**moving_average**(*y: ndarray*, *n: int*) → ndarray

Compute moving average.

Parameters

- **y** (*numpy.ndarray*) – Input array
- **n** (*int*) – Window size

Returns
Moving average

Return type
np.ndarray

cdl.algorithms.signal.**derivative**(*x: ndarray*, *y: ndarray*) → ndarray

Compute numerical derivative.

Parameters

- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data

Returns
Numerical derivative

Return type
np.ndarray

cdl.algorithms.signal.**normalize**(*yin: ndarray*, *parameter: str = 'maximum'*) → ndarray

Normalize input array to a given parameter.

Parameters

- **yin** (*numpy.ndarray*) – Input array
- **parameter** (*str | None*) – Normalization parameter. Defaults to “maximum”. Supported values: ‘maximum’, ‘amplitude’, ‘sum’, ‘energy’

Returns
Normalized array

Return type
np.ndarray

cdl.algorithms.signal.**xy_fft**(*x: ndarray*, *y: ndarray*, *shift: bool = True*) → tuple[ndarray, ndarray]

Compute FFT on X,Y data.

Parameters

- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data
- **shift** (*bool | None*) – Shift the zero frequency to the center of the spectrum. Defaults to True.

Returns
X,Y data

Return type
tuple[np.ndarray, np.ndarray]

cdl.algorithms.signal.**xy_ifft**(*x: ndarray*, *y: ndarray*, *shift: bool = True*) → tuple[ndarray, ndarray]

Compute iFFT on X,Y data.

Parameters

- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data
- **shift** (*bool | None*) – Shift the zero frequency to the center of the spectrum. Defaults to True.

Returns
X,Y data

Return type
tuple[np.ndarray, np.ndarray]

cdl.algorithms.signal.**peak_indexes**(*y*, *thres: float = 0.3*, *min_dist: int = 1*, *thres_abs: bool = False*) → ndarray

Peak detection routine.

Finds the numeric index of the peaks in *y* by taking its first order difference. By using *thres* and *min_dist* parameters, it is possible to reduce the number of detected peaks. *y* must be signed.

Parameters

- **y** (*ndarray (signed)*) – 1D amplitude data to search for peaks.
- **thres** (*float between [0., 1.]*) – Normalized threshold. Only the peaks with amplitude higher than the threshold will be detected.
- **min_dist** (*int*) – Minimum distance between each detected peak. The peak with the highest amplitude is preferred to satisfy this constraint.
- **thres_abs** (*boolean*) – If True, the thres value will be interpreted as an absolute value, instead of a normalized threshold.

Returns
Array containing the numeric indexes of the peaks that were detected

Return type
ndarray

cdl.algorithms.signal.**xpeak**(*x: ndarray*, *y: ndarray*) → float

Return default peak X-position (assuming a single peak).

Parameters

- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data

Returns
Peak X-position

Return type
float

cdl.algorithms.signal.**interpolate**(*x: ndarray*, *y: ndarray*, *xnew: ndarray*, *method: str*, *fill_value: float | None = None*) → ndarray

Interpolate data.

Parameters

- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data
- **xnew** (*numpy.ndarray*) – New X data
- **method** (*str*) – Interpolation method. Valid values are 'linear', 'spline', 'quadratic', 'cubic', 'barycentric', 'pchip'
- **fill_value** (*float | None*) – Fill value. Defaults to None. This value is used to fill in for requested points outside of the X data range. It is only used if the method argument is 'linear', 'cubic' or 'pchip'.

3.1.2 Image Processing Algorithms

cdl.algorithms.image.**scale_data_to_min_max**(*data: ndarray*, *zmin: float | int*, *zmax: float | int*) → ndarray

Scale array *data* to fit [zmin, zmax] dynamic range

Parameters

- **data** – Input data
- **zmin** – Minimum value of output data
- **zmax** – Maximum value of output data

Returns
Scaled data

cdl.algorithms.image.**z_fft**(*z: ndarray*, *shift: bool = True*) → ndarray

Compute FFT of complex array *z*

Parameters

- **z** – Input data
- **shift** – Shift zero frequency to center (default: True)

Returns
FFT of input data

cdl.algorithms.image.**z_ifft**(*z: ndarray*, *shift: bool = True*) → ndarray

Compute inverse FFT of complex array *z*

Parameters

- **z** – Input data
- **shift** – Shift zero frequency to center (default: True)

Returns
Inverse FFT of input data

cdl.algorithms.image.**binning**(*data: ndarray*, *binning_x: int*, *binning_y: int*, *operation: str*, *dtype=None*) → ndarray

Perform image pixel binning

Parameters

- **data** – Input data
- **binning_x** – Binning factor along x-axis
- **binning_y** – Binning factor along y-axis

- **operation** – Binning operation (sum, average, median, min, max)
- **dtype** – Output data type (default: None, i.e. same as input)

Returns
Binned data

cdl.algorithms.image.**flatfield**(*rawdata: ndarray*, *flatdata: ndarray*, *threshold: float | None = None*) → ndarray

Compute flat-field correction

Parameters

- **rawdata** – Raw data
- **flatdata** – Flat-field data
- **threshold** – Threshold for flat-field correction (default: None)

Returns
Flat-field corrected data

cdl.algorithms.image.**get_centroid_fourier**(*data: ndarray*) → tuple[float, float]

Return image centroid using Fourier algorithm

Parameters
data – Input data

Returns
Centroid coordinates (row, col)

cdl.algorithms.image.**get_absolute_level**(*data: ndarray*, *level: float*) → float

Return absolute level

Parameters

- **data** – Input data
- **level** – Relative level (0.0 to 1.0)

Returns
Absolute level

cdl.algorithms.image.**get_enclosing_circle**(*data: ndarray*, *level: float = 0.5*) → tuple[int, int, float]

Return (x, y, radius) for the circle contour enclosing image values above threshold relative level (.5 means FWHM)

Parameters

- **data** – Input data
- **level** – Relative level (default: 0.5)

Returns
A tuple (x, y, radius)

Raises
ValueError – No contour was found

cdl.algorithms.image.**get_radial_profile**(*data: ndarray*, *center: tuple[int, int]*) → tuple[ndarray, ndarray]

Return radial profile of image data

Parameters

- **data** – Input data (2D array)

- **center** – Coordinates of the center of the profile (x, y)

Returns
Radial profile (X, Y) where X is the distance from the center (1D array) and Y is the average value of pixels at this distance (1D array)

cdl.algorithms.image.**distance_matrix**(*coords: list*) → ndarray

Return distance matrix from coords

Parameters
coords – List of coordinates

Returns
Distance matrix

cdl.algorithms.image.**get_2d_peaks_coords**(*data: ndarray*, *size: int | None = None*, *level: float = 0.5*) → 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.

Parameters

- **data** – Input data
- **size** – Neighborhood size (default: None)
- **level** – Relative level (default: 0.5)

Returns
Coordinates of peaks

cdl.algorithms.image.**get_contour_shapes**(*data: ndarray | MaskedArray*, *shape: str = 'ellipse'*, *level: float = 0.5*) → ndarray

Find iso-valued contours in a 2D array, above relative level (.5 means FWHM), then fit contours with shape ('ellipse' or 'circle')

Parameters

- **data** – Input data
- **shape** – Shape to fit. Valid values: 'circle', 'ellipse', 'polygon'. (default: 'ellipse')
- **level** – Relative level (default: 0.5)

Returns
Coordinates of shapes

cdl.algorithms.image.**get_hough_circle_peaks**(*data: ndarray*, *min_radius: float | None = None*, *max_radius: float | None = None*, *nb_radius: int | None = None*, *min_distance: int = 1*) → ndarray

Detect peaks in image from circle Hough transform, return circle coordinates.

Parameters

- **data** – Input data
- **min_radius** – Minimum radius (default: None)
- **max_radius** – Maximum radius (default: None)
- **nb_radius** – Number of radii (default: None)

- **min_distance** – Minimum distance between circles (default: 1)

Returns
Coordinates of circles

cdl.algorithms.image.**find_blobs_dog**(*data: ndarray*, *min_sigma: float = 1*, *max_sigma: float = 30*, *overlap: float = 0.5*, *threshold_rel: float = 0.2*, *exclude_border: bool = True*) → ndarray

Finds blobs in the given grayscale image using the Difference of Gaussians (DoG) method.

Parameters

- **data** – The grayscale input image.
- **min_sigma** – The minimum blob radius in pixels.
- **max_sigma** – The maximum blob radius in pixels.
- **overlap** – The minimum overlap between two blobs in pixels. For instance, if two blobs are detected with radii of 10 and 12 respectively, and the `overlap` is set to 0.5, then the area of the smaller blob will be ignored and only the area of the larger blob will be returned.
- **threshold_rel** – The absolute lower bound for scale space maxima. Local maxima smaller than `threshold_rel` are ignored. Reduce this to detect blobs with less intensities.
- **exclude_border** – If `True`, exclude blobs from detection if they are too close to the border of the image. Border size is `min_sigma`.

Returns
Coordinates of blobs

cdl.algorithms.image.**find_blobs_doh**(*data: ndarray*, *min_sigma: float = 1*, *max_sigma: float = 30*, *overlap: float = 0.5*, *log_scale: bool = False*, *threshold_rel: float = 0.2*) → ndarray

Finds blobs in the given grayscale image using the Determinant of Hessian (DoH) method.

Parameters

- **data** – The grayscale input image.
- **min_sigma** – The minimum blob radius in pixels.
- **max_sigma** – The maximum blob radius in pixels.
- **overlap** – The minimum overlap between two blobs in pixels. For instance, if two blobs are detected with radii of 10 and 12 respectively, and the `overlap` is set to 0.5, then the area of the smaller blob will be ignored and only the area of the larger blob will be returned.
- **log_scale** – If `True`, the radius of each blob is returned as `sqrt(sigma)` for each detected blob.
- **threshold_rel** – The absolute lower bound for scale space maxima. Local maxima smaller than `threshold_rel` are ignored. Reduce this to detect blobs with less intensities.

Returns
Coordinates of blobs

cdl.algorithms.image.**find_blobs_log**(*data: ndarray*, *min_sigma: float = 1*, *max_sigma: float = 30*, *overlap: float = 0.5*, *log_scale: bool = False*, *threshold_rel: float = 0.2*, *exclude_border: bool = True*) → ndarray

Finds blobs in the given grayscale image using the Laplacian of Gaussian (LoG) method.

Parameters

- **data** – The grayscale input image.
- **min_sigma** – The minimum blob radius in pixels.
- **max_sigma** – The maximum blob radius in pixels.
- **overlap** – The minimum overlap between two blobs in pixels. For instance, if two blobs are detected with radii of 10 and 12 respectively, and the `overlap` is set to 0.5, then the area of the smaller blob will be ignored and only the area of the larger blob will be returned.
- **log_scale** – If `True`, the radius of each blob is returned as `sqrt(sigma)` for each detected blob.
- **threshold_rel** – The absolute lower bound for scale space maxima. Local maxima smaller than `threshold_rel` are ignored. Reduce this to detect blobs with less intensities.
- **exclude_border** – If `True`, exclude blobs from detection if they are too close to the border of the image. Border size is `min_sigma`.

Returns
Coordinates of blobs

cdl.algorithms.image.**remove_overlapping_disks**(*coords: ndarray*) → ndarray

Remove overlapping disks among coordinates

Parameters
coords – The coordinates of the disks

Returns
The coordinates of the disks with overlapping disks removed

cdl.algorithms.image.**find_blobs_opencv**(*data: ndarray*, *min_threshold: float | None = None*, *max_threshold: float | None = None*, *min_repeatability: int | None = None*, *min_dist_between_blobs: float | None = None*, *filter_by_color: bool | None = None*, *blob_color: int | None = None*, *filter_by_area: bool | None = None*, *min_area: float | None = None*, *max_area: float | None = None*, *filter_by_circularity: bool | None = None*, *min_circularity: float | None = None*, *max_circularity: float | None = None*, *filter_by_inertia: bool | None = None*, *min_inertia_ratio: float | None = None*, *max_inertia_ratio: float | None = None*, *filter_by_convexity: bool | None = None*, *min_convexity: float | None = None*, *max_convexity: float | None = None*) → ndarray

Finds blobs in the given grayscale image using OpenCV's SimpleBlobDetector.

Parameters

- **data** – The grayscale input image.
- **min_threshold** – The minimum blob intensity.
- **max_threshold** – The maximum blob intensity.
- **min_repeatability** – The minimum number of times a blob is detected before it is reported.
- **min_dist_between_blobs** – The minimum distance between blobs.
- **filter_by_color** – If `True`, blobs are filtered by color.
- **blob_color** – The color of the blobs to filter by.
- **filter_by_area** – If `True`, blobs are filtered by area.

- **min_area** – The minimum blob area.
- **max_area** – The maximum blob area.
- **filter_by_circularity** – If True, blobs are filtered by circularity.
- **min_circularity** – The minimum blob circularity.
- **max_circularity** – The maximum blob circularity.
- **filter_by_inertia** – If True, blobs are filtered by inertia.
- **min_inertia_ratio** – The minimum blob inertia ratio.
- **max_inertia_ratio** – The maximum blob inertia ratio.
- **filter_by_convexity** – If True, blobs are filtered by convexity.
- **min_convexity** – The minimum blob convexity.
- **max_convexity** – The maximum blob convexity.

Returns
Coordinates of blobs

3.1.3 Data Type Conversion Algorithms

cdl.algorithms.datatypes.**is_integer_dtype**(*dtype: dtype*) → bool

Return True if data type is an integer type

Parameters
dtype – Data type to check

Returns
True if data type is an integer type

cdl.algorithms.datatypes.**is_complex_dtype**(*dtype: dtype*) → bool

Return True if data type is a complex type

Parameters
dtype – Data type to check

Returns
True if data type is a complex type

3.1.4 Coordinate Conversion Algorithms

cdl.algorithms.coordinates.**circle_center_radius_to_diameter**(*xc: float*, *yc: float*, *r: float*) → tuple[float, float, float, float]

Convert circle center and radius to X diameter coordinates

Parameters

- **xc** – Circle center X coordinate
- **yc** – Circle center Y coordinate
- **r** – Circle radius

Returns
Circle X diameter coordinates

Return type
tuple

cdl.algorithms.coordinates.**ellipse_center_axes_angle_to_diameters**(*xc: float*, *yc: float*, *a: float*, *b: float*, *theta: float*) → tuple[float, float, float, float, float, float, float, float]

Convert ellipse center, axes and angle to X/Y diameters coordinates

Parameters

- **xc** – Ellipse center X coordinate
- **yc** – Ellipse center Y coordinate
- **a** – Ellipse half larger axis
- **b** – Ellipse half smaller axis
- **theta** – Ellipse angle

Returns
Ellipse X/Y diameters coordinates

3.1.5 Curve Fitting Algorithms

class cdl.algorithms.fit.**FitModel**

Curve fitting model base class

abstract classmethod func(*x*, *amp*, *sigma*, *x0*, *y0*)

Return fitting function

classmethod get_amp_from_amplitude(*amplitude*, *sigma*)

Return amp from function amplitude and sigma

classmethod amplitude(*amp*, *sigma*)

Return function amplitude

abstract classmethod fwhm(*amp*, *sigma*)

Return function FWHM

classmethod half_max_segment(*amp*, *sigma*, *x0*, *y0*)

Return segment coordinates for y=half-maximum intersection

class cdl.algorithms.fit.**GaussianModel**

1-dimensional Gaussian fit model

classmethod func(*x*, *amp*, *sigma*, *x0*, *y0*)

Return fitting function

classmethod get_amp_from_amplitude(*amplitude*, *sigma*)

Return amp from function amplitude and sigma

classmethod amplitude(*amp*, *sigma*)

Return function amplitude

classmethod fwhm(*amp*, *sigma*)

Return function FWHM

class cdl.algorithms.fit.**LorentzianModel**

1-dimensional Lorentzian fit model

classmethod func(*x*, *amp*, *sigma*, *x0*, *y0*)

Return fitting function

classmethod get_amp_from_amplitude(*amplitude*, *sigma*)

Return amp from function amplitude and sigma

classmethod amplitude(*amp*, *sigma*)

Return function amplitude

classmethod fwhm(*amp*, *sigma*)

Return function FWHM

class cdl.algorithms.fit.**VoigtModel**

1-dimensional Voigt fit model

classmethod func(*x*, *amp*, *sigma*, *x0*, *y0*)

Return fitting function

classmethod fwhm(*amp*, *sigma*)

Return function FWHM

3.2 Parameters (cdl.param)

The *cdl.param* module aims at providing all the dataset parameters that are used by the *cdl.core.computation* and cdl.core.gui.processor packages.

Those datasets (guidata.dataset.datatypes.Dataset subclasses) are defined in other modules:

- *cdl.core.computation.base*
- *cdl.core.computation.image*
- *cdl.core.computation.signal*

The *cdl.param* module is thus a convenient way to import all the sets of parameters at once.

As a matter of fact, the following import statement is equivalent to the previous one:

```
# Original import statement
from cdl.core.computation.base import MovingAverageParam
from cdl.core.computation.signal import PolynomialFitParam
from cdl.core.computation.image.exposure import EqualizeHistParam

# Equivalent import statement
from cdl.param import MovingAverageParam, PolynomialFitParam, EqualizeHistParam
```

3.2.1 Common parameters

class cdl.param.**ClipParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Data clipping parameters

class cdl.param.**FFTParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

FFT parameters

class cdl.param.**GaussianParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Gaussian filter parameters

class cdl.param.**MovingAverageParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Moving average parameters

class cdl.param.**MovingMedianParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Moving median parameters

class cdl.param.**ROIDataParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

ROI Editor data

classmethod create(*roidata: ndarray | None = None*, *singleobj: bool | None = None*)

Create ROIDataParam instance

property is_empty: bool

Return True if there is no ROI

class cdl.param.**ThresholdParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Threshold parameters

3.2.2 Signal parameters

class cdl.param.**DataTypeSParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Convert signal data type parameters

class cdl.param.**FWHMParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

FWHM parameters

class cdl.param.**NormalizeYParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Normalize parameters

class cdl.param.**PeakDetectionParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Peak detection parameters

class cdl.param.**PolynomialFitParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Polynomial fitting parameters

class cdl.param.**XYCalibrateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Signal calibration parameters

class cdl.param.**InterpolationParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Interpolation parameters

class cdl.param.**ResamplingParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Resample parameters

class cdl.param.**DetrendingParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Detrending parameters

3.2.3 Image parameters

Base image parameters

class cdl.param.**AverageProfileParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Average horizontal or vertical profile parameters

class cdl.param.**RadialProfileParam**(**args*, ***kwargs*)

Radial profile parameters

update_from_image(*obj:* ImageObj) → None

Update parameters from image

choice_callback(*item*, *value*)

Callback for choice item

class cdl.param.**BinningParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Binning parameters

class cdl.param.**ButterworthParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Butterworth filter parameters

class cdl.param.**DataTypeIParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Convert image data type parameters

class cdl.param.**FlatFieldParam**(*title=None*, *comment=None*, *icon=''*)

Flat-field parameters

class cdl.param.**GridParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Grid parameters

class cdl.param.**HoughCircleParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Circle Hough transform parameters

class cdl.param.**LogP1Param**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Log10 parameters

class cdl.param.**ProfileParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Horizontal or vertical profile parameters

class cdl.param.**ResizeParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Resize parameters

class cdl.param.**RotateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Rotate parameters

class cdl.param.**ZCalibrateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Image linear calibration parameters

Exposure correction parameters

class cdl.param.**AdjustGammaParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Gamma adjustment parameters

class cdl.param.**AdjustLogParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Logarithmic adjustment parameters

class cdl.param.**AdjustSigmoidParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Sigmoid adjustment parameters

class cdl.param.**EqualizeAdaptHistParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Adaptive histogram equalization parameters

class cdl.param.**EqualizeHistParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Histogram equalization parameters

class cdl.param.**RescaleIntensityParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Intensity rescaling parameters

Restoration parameters

class cdl.param.**DenoiseBilateralParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Bilateral filter denoising parameters

class cdl.param.**DenoiseTVParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Total Variation denoising parameters

class cdl.param.**DenoiseWaveletParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Wavelet denoising parameters

Morphological parameters

class cdl.param.**MorphologyParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

White Top-Hat parameters

Edge detection parameters

class cdl.param.**CannyParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Canny filter parameters

Detection parameters

class cdl.param.**BlobDOGParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Difference of Gaussian method

class cdl.param.**BlobDOHParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Determinant of Hessian method

class cdl.param.**BlobLOGParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Laplacian of Gaussian method

class cdl.param.**BlobOpenCVParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using OpenCV

class cdl.param.**ContourShapeParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Contour shape parameters

class cdl.param.**Peak2DDetectionParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Peak detection parameters

3.3 Object model (cdl.obj)

The *cdl.obj* module aims at providing all the necessary classes and functions to create and manipulate DataLab signal and image objects.

Those classes and functions are defined in other modules:

- cdl.core.model.base
- cdl.core.model.image
- cdl.core.model.signal
- cdl.core.io

The *cdl.obj* module is thus a convenient way to import all the objects at once. As a matter of fact, the following import statement is equivalent to the previous one:

```
# Original import statement
from cdl.core.model.signal import SignalObj
from cdl.core.model.image import ImageObj

# Equivalent import statement
from cdl.obj import SignalObj, ImageObj
```

3.3.1 Common objects

class cdl.obj.**ResultShape**(*shapetype:* ShapeTypes, *array:* *ndarray*, *label:* *str* = '')

Object representing a geometrical shape serializable in signal/image metadata.

Result *array* is a NumPy 2-D array: each row is a result, optionnally associated to a ROI (first column value).

ROI index is starting at 0 (or is simply 0 if there is no ROI).

Parameters

- **shapetype** – shape type
- **array** – shape coordinates (multiple shapes: one shape per row), first column is ROI index (0 if there is no ROI)
- **label** – shape label

Raises

AssertionError – invalid argument

classmethod label_shapetype_from_key(*key:* *str*)

Return metadata shape label and shapetype from metadata key

classmethod from_metadata_entry(*key*, *value*) → *ResultShape* | None

Create metadata shape object from (key, value) metadata entry

classmethod match(*key*, *value*) → bool

Return True if metadata dict entry (key, value) is a metadata result

property key: str

Return metadata key associated to result

property shown_xlabels: tuple[str]

Return labels for result array columns

property shown_array: ndarray

Return array of shown results, i.e. including the complementary array

Returns
Array of shown results

add_to(*obj: BaseObj*)

Add metadata shape to object (signal/image)

merge_with(*obj: BaseObj*, *other_obj: BaseObj | None = None*)

Merge object resultshape with another's: obj <– other_obj or simply merge this resultshape with obj if other_obj is None

property data_colnb

Return raw data results column number

is_first_column_roi_index() → bool

Return True if first column is ROI index

property data

Return raw data (array without ROI informations)

check_array()

Check if array is valid

iterate_plot_items(*fmt: str*, *lbl: bool*, *option: str*) → Iterable

Iterate over metadata shape plot items.

Parameters

- **fmt** (*str*) – numeric format (e.g. "%.3f")
- **lbl** (*bool*) – if True, show shape labels
- **option** (*str*) – shape style option (e.g. "shape/drag")

Yields
PlotItem – plot item

create_plot_item(*args: ndarray*, *fmt: str*, *lbl: bool*, *option: str*)

Make plot item.

Parameters

- **args** (*numpy.ndarray*) – shape data
- **fmt** (*str*) – numeric format (e.g. "%.3f")
- **lbl** (*bool*) – if True, show shape labels
- **option** (*str*) – shape style option (e.g. "shape/drag")

Returns
plot item

Return type
PlotItem

make_marker_item(*args*, *fmt*)

Make marker item

class cdl.obj.**ShapeTypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Shape types for image metadata

class cdl.obj.**UniformRandomParam**(*title=None*, *comment=None*, *icon=''*)

Uniform-law random signal/image parameters

apply_integer_range(*vmin*, *vmax*)

Do something in case of integer min-max range

class cdl.obj.**NormalRandomParam**(*title=None*, *comment=None*, *icon=''*)

Normal-law random signal/image parameters

apply_integer_range(*vmin*, *vmax*)

Do something in case of integer min-max range

3.3.2 Signal model

class cdl.obj.**SignalObj**(*title=None*, *comment=None*, *icon=''*)

Signal object

regenerate_uuid()

Regenerate UUID

This method is used to regenerate UUID after loading the object from a file. This is required to avoid UUID conflicts when loading objects from file without clearing the workspace first.

copy(*title: str | None = None*, *dtype: dtype | None = None*) → *SignalObj*

Copy object.

Parameters

- **title** (*str*) – title
- **dtype** (*numpy.dtype*) – data type

Returns
copied object

Return type
SignalObj

set_data_type(*dtype: dtype*) → None

Change data type.

Parameters
dtype (*numpy.dtype*) – data type

set_xydata(*x: ndarray | list*, *y: ndarray | list*, *dx: ndarray | list | None = None*, *dy: ndarray | list | None = None*) → None

Set xy data

Parameters

- **x** (*numpy.ndarray*) – x data
- **y** (*numpy.ndarray*) – y data
- **dx** (*numpy.ndarray*) – dx data (optional: error bars)

- **dy** (*numpy.ndarray*) – dy data (optional: error bars)

property x: ndarray | None

Get x data

property y: ndarray | None

Get y data

property data: ndarray | None

Get y data

property dx: ndarray | None

Get dx data

property dy: ndarray | None

Get dy data

get_data(*roi_index: int | None = None*) → ndarray

Return original data (if ROI is not defined or *roi_index* is None), or ROI data (if both ROI and *roi_index* are defined).

Parameters
roi_index (*int*) – ROI index

Returns
data

Return type
numpy.ndarray

update_plot_item_parameters(*item: CurveItem*) → None

Update plot item parameters from object data/metadata

Takes into account a subset of plot item parameters. Those parameters may have been overriden by object metadata entries or other object data. The goal is to update the plot item accordingly.

This is *almost* the inverse operation of *update_metadata_from_plot_item*.

Parameters
item – plot item

update_metadata_from_plot_item(*item: CurveItem*) → None

Update metadata from plot item.

Takes into account a subset of plot item parameters. Those parameters may have been modified by the user through the plot item GUI. The goal is to update the metadata accordingly.

This is *almost* the inverse operation of *update_plot_item_parameters*.

Parameters
item – plot item

make_item(*update_from: CurveItem = None*) → CurveItem

Make plot item from data.

Parameters
update_from (*CurveItem*) – plot item to update from

Returns
plot item

Return type
CurveItem

update_item(*item: CurveItem*, *data_changed: bool = True*) → None

Update plot item from data.

Parameters

- **item** (*CurveItem*) – plot item
- **data_changed** (*bool*) – if True, data has changed

roi_coords_to_indexes(*coords: list*) → ndarray

Convert ROI coordinates to indexes.

Parameters
coords (*list*) – coordinates

Returns
indexes

Return type
numpy.ndarray

get_roi_param(*title: str*, **defaults*) → DataSet

Return ROI parameters dataset.

Parameters

- **title** (*str*) – title
- ***defaults** – default values

static params_to_roidata(*params: DataSetGroup*) → ndarray

Convert ROI dataset group to ROI array data.

Parameters
params (*DataSetGroup*) – ROI dataset group

Returns
ROI array data

Return type
numpy.ndarray

new_roi_item(*fmt: str*, *lbl: bool*, *editable: bool*)

Return a new ROI item from scratch

Parameters

- **fmt** (*str*) – format string
- **lbl** (*bool*) – if True, add label
- **editable** (*bool*) – if True, ROI is editable

iterate_roi_items(*fmt: str*, *lbl: bool*, *editable: bool = True*)

Make plot item representing a Region of Interest.

Parameters

- **fmt** (*str*) – format string
- **lbl** (*bool*) – if True, add label
- **editable** (*bool*) – if True, ROI is editable

Yields
PlotItem – plot item

add_label_with_title(*title: str | None = None*) → None

Add label with title annotation

Parameters
title (*str*) – title (if None, use signal title)

accept(*vis: object*) → None

Helper function that passes the visitor to the accept methods of all the items in this dataset

Parameters
vis (*object*) – visitor object

add_annotations_from_file(*filename: str*) → None

Add object annotations from file (JSON).

Parameters
filename (*str*) – filename

add_annotations_from_items(*items: list*) → None

Add object annotations (annotation plot items).

Parameters
items (*list*) – annotation plot items

add_resultshape(*label: str*, *shapetype: ShapeTypes*, *array: ndarray*, *param: DataSet | None = None*) → *ResultShape*

Add geometric shape as metadata entry, and return ResultShape instance.

Parameters

- **label** (*str*) – label
- **shapetype** (*ShapeTypes*) – shape type
- **array** (*numpy.ndarray*) – array
- **param** (*guidata.dataset.DataSet*) – parameters

Returns
result shape

Return type
ResultShape

property annotations: str

Get object annotations (JSON string describing annotation plot items)

check() → list[str]

Check the dataset item values

Returns
list of errors

Return type
list[str]

check_data()

Check if data is valid, raise an exception if that's not the case

Raises
TypeError – if data type is not supported

classmethod create(***kwargs*) → DataSet

Create a new instance of the DataSet class

Parameters
kwargs – keyword arguments to set the DataItem values

Returns
DataSet instance

deserialize(*reader: HDF5Reader | JSONReader | INIReader*) → None

Deserialize the dataset

Parameters
reader (*HDF5Reader | JSONReader | INIReader*) – reader object

edit(*parent: QWidget | None = None*, *apply: Callable | None = None*, *wordwrap: bool = True*, *size: QSize | tuple[int, int] | None = None*) → DataSetEditDialog

Open a dialog box to edit data set

Parameters

- **parent** – parent widget (default is None, meaning no parent)
- **apply** – apply callback (default is None)
- **wordwrap** – if True, comment text is wordwrapped
- **size** – dialog size (QSize object or integer tuple (width, height))

export_metadata_to_file(*filename: str*) → None

Export object metadata to file (JSON).

Parameters
filename (*str*) – filename

get_comment() → str | None

Return data set comment

Returns
comment

Return type
str | None

get_icon() → str | None

Return data set icon

Returns
icon

Return type
str | None

get_items(*copy=False*) → list[DataItem]

Returns all the DataItem objects from the DataSet instance. Ignore private items that have a name starting with an underscore (e.g. '_private_item = ...')

Parameters

- **copy** – If True, deepcopy the DataItem list, else return the original.
- **False.** (*Defaults to*) –

Returns
description

`get_metadata_option`(*name: str*) → Any

Return metadata option value

A metadata option is a metadata entry starting with an underscore. It is a way to store application-specific options in object metadata.

Parameters
name (*str*) – option name

Returns
Option value

Valid option names:
'format': format string (str) 'showlabel': show label (bool)

`get_title`() → str

Return data set title

Returns
title

Return type
str

`classmethod get_valid_dtypenames`() → list[str]

Get valid data type names

Returns
Valid data type names supported by this class

`import_metadata_from_file`(*filename: str*) → None

Import object metadata from file (JSON).

Parameters
filename (*str*) – filename

`iterate_resultshapes`()

Iterate over object result shapes.

Yields
ResultShape – result shape

`iterate_roi_indexes`()

Iterate over object ROI indexes ([0] if there is no ROI)

`iterate_shape_items`(*editable: bool = False*)

Iterate over computing items encoded in metadata (if any).

Parameters
editable (*bool*) – if True, ROI is editable

Yields
PlotItem – plot item

`metadata_to_html`() → str

Convert metadata to human-readable string.

Returns
HTML string

Return type
str

property number: int

Return object number (used for short ID)

read_config(*conf: UserConfig*, *section: str*, *option: str*) → None

Read configuration from a UserConfig instance

Parameters

- **conf** (*UserConfig*) – UserConfig instance
- **section** (*str*) – section name
- **option** (*str*) – option name

remove_all_shapes() → None

Remove metadata shapes and ROIs

reset_metadata_to_defaults() → None

Reset metadata to default values

property roi: ndarray | None

Return object regions of interest array (one ROI per line).

Returns
regions of interest array

Return type
numpy.ndarray

roi_has_changed() → bool

Return True if ROI has changed since last call to this method.

The first call to this method will return True if ROI has not yet been set, or if ROI has been set and has changed since the last call to this method. The next call to this method will always return False if ROI has not changed in the meantime.

Returns
True if ROI has changed

Return type
bool

roidata_to_params(*roidata: ndarray*) → DataSetGroup

Convert ROI array data to ROI dataset group.

Parameters
roidata (*numpy.ndarray*) – ROI array data

Returns
ROI dataset group

Return type
DataSetGroup

serialize(*writer: HDF5Writer | JSONWriter | INIWriter*) → None

Serialize the dataset

Parameters
writer (*HDF5Writer | JSONWriter | INIWriter*) – writer object

set_annotations_from_file(*filename: str*) → None

Set object annotations from file (JSON).

Parameters
filename (*str*) – filename

set_defaults() → None

Set default values

classmethod set_global_prop(*realm: str*, ***kwargs*) → None

Set global properties for all data items in the dataset

Parameters

- **realm** (*str*) – realm name
- **kwargs** (*dict*) – properties to set

set_metadata_option(*name: str*, *value: Any*) → None

Set metadata option value

A metadata option is a metadata entry starting with an underscore. It is a way to store application-specific options in object metadata.

Parameters

- **name** (*str*) – option name
- **value** (*Any*) – option value

Valid option names:
'format': format string (str) 'showlabel': show label (bool)

property short_id

Short object ID

text_edit() → None

Edit data set with text input only

to_string(*debug: bool | None = False*, *indent: str | None = None*, *align: bool | None = False*, *show_hidden: bool | None = True*) → str

Return readable string representation of the data set If debug is True, add more details on data items

Parameters

- **debug** (*bool*) – if True, add more details on data items
- **indent** (*str*) – indentation string (default is None, meaning no indentation)
- **align** (*bool*) – if True, align data items (default is False)
- **show_hidden** (*bool*) – if True, show hidden data items (default is True)

Returns
string representation of the data set

Return type
str

transform_shapes(*orig*, *func*, *param=None*)

Apply transform function to result shape / annotations coordinates.

Parameters

- **orig** (*BaseObj*) – original object
- **func** (*callable*) – transform function
- **param** (*object*) – transform function parameter

update_metadata_view_settings() → None

Update metadata view settings from Conf.view

update_resultshapes_from(*other: BaseObj*) → None

Update geometric shape from another object (merge metadata).

Parameters
other – other object, from which to update this object

view(*parent: QWidget | None = None*, *wordwrap: bool = True*, *size: QSize | tuple[int, int] | None = None*) → None

Open a dialog box to view data set

Parameters

- **parent** – parent widget (default is None, meaning no parent)
- **wordwrap** – if True, comment text is wordwrapped
- **size** – dialog size (QSize object or integer tuple (width, height))

write_config(*conf: UserConfig*, *section: str*, *option: str*) → None

Write configuration to a UserConfig instance

Parameters

- **conf** (*UserConfig*) – UserConfig instance
- **section** (*str*) – section name
- **option** (*str*) – option name

cdl.obj.**read_signal**(*filename: str*) → *SignalObj*

Read a signal from a file.

Parameters
filename (*str*) – File name.

Returns
Signal.

Return type
Signal

cdl.obj.**create_signal**(*title: str*, *x: ndarray | None = None*, *y: ndarray | None = None*, *dx: ndarray | None = None*, *dy: ndarray | None = None*, *metadata: dict | None = None*, *units: tuple | None = None*, *labels: tuple | None = None*) → *SignalObj*

Create a new Signal object.

Parameters

- **title** (*str*) – signal title
- **x** (*numpy.ndarray*) – X data
- **y** (*numpy.ndarray*) – Y data
- **dx** (*numpy.ndarray*) – dX data (optional: error bars)
- **dy** (*numpy.ndarray*) – dY data (optional: error bars)
- **metadata** (*dict*) – signal metadata
- **units** (*tuple*) – X, Y units (tuple of strings)
- **labels** (*tuple*) – X, Y labels (tuple of strings)

Returns
signal object

Return type
SignalObj

cdl.obj.**create_signal_from_param**(*newparam: NewSignalParam*, *addparam: gds.DataSet | None = None*, *edit: bool = False*, *parent: QW.QWidget | None = None*) → *SignalObj* | None

Create a new Signal object from a dialog box.

Parameters

- **newparam** (*NewSignalParam*) – new signal parameters
- **addparam** (*guidata.dataset.DataSet*) – additional parameters
- **edit** (*bool*) – Open a dialog box to edit parameters (default: False)
- **parent** (*QWidget*) – parent widget

Returns
signal object or None if canceled

Return type
SignalObj

cdl.obj.**new_signal_param**(*title: str | None = None*, *stype: str | None = None*, *xmin: float | None = None*, *xmax: float | None = None*, *size: int | None = None*) → *NewSignalParam*

Create a new Signal dataset instance.

Parameters

- **title** (*str*) – dataset title (default: None, uses default title)
- **stype** (*str*) – signal type (default: None, uses default type)
- **xmin** (*float*) – X min (default: None, uses default value)
- **xmax** (*float*) – X max (default: None, uses default value)
- **size** (*int*) – signal size (default: None, uses default value)

Returns
new signal dataset instance

Return type
NewSignalParam

class cdl.obj.**SignalTypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Signal types

class cdl.obj.**NewSignalParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

New signal dataset

class cdl.obj.**GaussLorentzVoigtParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Parameters for Gaussian and Lorentzian functions

class cdl.obj.**StepParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Parameters for step function

class cdl.obj.**PeriodicParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Parameters for periodic functions

get_frequency_in_hz()

Return frequency in Hz

3.3.3 Image model

class cdl.obj.**ImageObj**(*title=None*, *comment=None*, *icon=''*)

Image object

regenerate_uuid()

Regenerate UUID

This method is used to regenerate UUID after loading the object from a file. This is required to avoid UUID conflicts when loading objects from file without clearing the workspace first.

set_metadata_from(*obj: Mapping | dict*) → None

Set metadata from object: dict-like (only string keys are considered) or any other object (iterating over supported attributes)

Parameters
obj (*Mapping | dict*) – object

property dicom_template

Get DICOM template

property xc: float

Return image center X-axis coordinate

property yc: float

Return image center Y-axis coordinate

get_data(*roi_index: int | None = None*) → ndarray

Return original data (if ROI is not defined or *roi_index* is None), or ROI data (if both ROI and *roi_index* are defined).

Parameters
roi_index (*int*) – ROI index

Returns
masked data

Return type
numpy.ndarray

copy(*title: str | None = None*, *dtype: dtype | None = None*) → *ImageObj*

Copy object.

Parameters

- **title** (*str*) – title
- **dtype** (*numpy.dtype*) – data type

Returns
copied object

Return type
ImageObj

set_data_type(*dtype: dtype*) → None

Change data type.

Parameters
dtype (*numpy.dtype*) – data type

update_plot_item_parameters(*item: MaskedImageItem*) → None

Update plot item parameters from object data/metadata

Takes into account a subset of plot item parameters. Those parameters may have been overriden by object metadata entries or other object data. The goal is to update the plot item accordingly.

This is *almost* the inverse operation of *update_metadata_from_plot_item*.

Parameters
item – plot item

update_metadata_from_plot_item(*item: MaskedImageItem*) → None

Update metadata from plot item.

Takes into account a subset of plot item parameters. Those parameters may have been modified by the user through the plot item GUI. The goal is to update the metadata accordingly.

This is *almost* the inverse operation of *update_plot_item_parameters*.

Parameters
item – plot item

make_item(*update_from: MaskedImageItem | None = None*) → MaskedImageItem

Make plot item from data.

Parameters
update_from (*MaskedImageItem | None*) – update from plot item

Returns
plot item

Return type
MaskedImageItem

update_item(*item: MaskedImageItem*, *data_changed: bool = True*) → None

Update plot item from data.

Parameters

- **item** (*MaskedImageItem*) – plot item
- **data_changed** (*bool*) – if True, data has changed

get_roi_param(*title*, **defaults*) → DataSet

Return ROI parameters dataset.

Parameters

- **title** (*str*) – title
- ***defaults** – default values

static params_to_roidata(*params: DataSetGroup*) → ndarray | None

Convert ROI dataset group to ROI array data.

Parameters
params (*DataSetGroup*) – ROI dataset group

Returns
ROI array data

Return type
numpy.ndarray

new_roi_item(*fmt: str*, *lbl: bool*, *editable: bool*, *geometry:* RoiDataGeometries) → MaskedImageItem

Return a new ROI item from scratch

Parameters

- **fmt** (*str*) – format string
- **lbl** (*bool*) – if True, add label
- **editable** (*bool*) – if True, ROI is editable
- **geometry** (RoiDataGeometries) – ROI geometry

roi_coords_to_indexes(*coords: list*) → ndarray

Convert ROI coordinates to indexes.

Parameters
coords (*list*) – coordinates

Returns
indexes

Return type
numpy.ndarray

iterate_roi_items(*fmt: str*, *lbl: bool*, *editable: bool = True*) → Iterator

Make plot item representing a Region of Interest.

Parameters

- **fmt** (*str*) – format string
- **lbl** (*bool*) – if True, add label
- **editable** (*bool*) – if True, ROI is editable

Yields
PlotItem – plot item

property maskdata: ndarray

Return masked data (areas outside defined regions of interest)

Returns
masked data

Return type
numpy.ndarray

get_masked_view() → MaskedArray

Return masked view for data

Returns
Masked view

invalidate_maskdata_cache() → None

Invalidate mask data cache: force to rebuild it

add_label_with_title(*title: str | None = None*) → None

Add label with title annotation

Parameters
title (*str*) – title (if None, use image title)

accept(*vis: object*) → None

Helper function that passes the visitor to the accept methods of all the items in this dataset

Parameters
vis (*object*) – visitor object

add_annotations_from_file(*filename: str*) → None

Add object annotations from file (JSON).

Parameters
filename (*str*) – filename

add_annotations_from_items(*items: list*) → None

Add object annotations (annotation plot items).

Parameters
items (*list*) – annotation plot items

add_resultshape(*label: str*, *shapetype:* ShapeTypes, *array: ndarray*, *param: DataSet | None = None*) → *ResultShape*

Add geometric shape as metadata entry, and return ResultShape instance.

Parameters

- **label** (*str*) – label
- **shapetype** (ShapeTypes) – shape type
- **array** (*numpy.ndarray*) – array
- **param** (*guidata.dataset.DataSet*) – parameters

Returns
result shape

Return type
ResultShape

property annotations: str

Get object annotations (JSON string describing annotation plot items)

check() → list[str]

Check the dataset item values

Returns
list of errors

Return type
list[str]

check_data()

Check if data is valid, raise an exception if that's not the case

Raises
TypeError – if data type is not supported

classmethod create(**kwargs) → DataSet

Create a new instance of the DataSet class

Parameters
kwargs – keyword arguments to set the DataItem values

Returns
DataSet instance

deserialize(*reader: HDF5Reader | JSONReader | INIReader*) → None

Deserialize the dataset

Parameters
reader (*HDF5Reader | JSONReader | INIReader*) – reader object

edit(*parent: QWidget | None = None*, *apply: Callable | None = None*, *wordwrap: bool = True*, *size: QSize | tuple[int, int] | None = None*) → DataSetEditDialog

Open a dialog box to edit data set

Parameters

- **parent** – parent widget (default is None, meaning no parent)
- **apply** – apply callback (default is None)
- **wordwrap** – if True, comment text is wordwrapped
- **size** – dialog size (QSize object or integer tuple (width, height))

export_metadata_to_file(*filename: str*) → None

Export object metadata to file (JSON).

Parameters
filename (*str*) – filename

get_comment() → str | None

Return data set comment

Returns
comment

Return type
str | None

`get_icon()` → str | None

Return data set icon

Returns
icon

Return type
str | None

`get_items`(*copy=False*) → list[DataItem]

Returns all the DataItem objects from the DataSet instance. Ignore private items that have a name starting with an underscore (e.g. '_private_item = ...')

Parameters

- **copy** – If True, deepcopy the DataItem list, else return the original.
- **False.** (*Defaults to*) –

Returns
description

`get_metadata_option`(*name: str*) → Any

Return metadata option value

A metadata option is a metadata entry starting with an underscore. It is a way to store application-specific options in object metadata.

Parameters
name (*str*) – option name

Returns
Option value

Valid option names:
'format': format string (str) 'showlabel': show label (bool)

`get_title()` → str

Return data set title

Returns
title

Return type
str

`classmethod get_valid_dtypenames()` → list[str]

Get valid data type names

Returns
Valid data type names supported by this class

`import_metadata_from_file`(*filename: str*) → None

Import object metadata from file (JSON).

Parameters
filename (*str*) – filename

iterate_resultshapes()

Iterate over object result shapes.

Yields
ResultShape – result shape

iterate_roi_indexes()

Iterate over object ROI indexes ([0] if there is no ROI)

iterate_shape_items(*editable: bool = False*)

Iterate over computing items encoded in metadata (if any).

Parameters
editable (*bool*) – if True, ROI is editable

Yields
PlotItem – plot item

metadata_to_html() → str

Convert metadata to human-readable string.

Returns
HTML string

Return type
str

property number: int

Return object number (used for short ID)

read_config(*conf: UserConfig*, *section: str*, *option: str*) → None

Read configuration from a UserConfig instance

Parameters

- **conf** (*UserConfig*) – UserConfig instance
- **section** (*str*) – section name
- **option** (*str*) – option name

remove_all_shapes() → None

Remove metadata shapes and ROIs

reset_metadata_to_defaults() → None

Reset metadata to default values

property roi: ndarray | None

Return object regions of interest array (one ROI per line).

Returns
regions of interest array

Return type
numpy.ndarray

roi_has_changed() → bool

Return True if ROI has changed since last call to this method.

The first call to this method will return True if ROI has not yet been set, or if ROI has been set and has changed since the last call to this method. The next call to this method will always return False if ROI has not changed in the meantime.

Returns
True if ROI has changed

Return type
bool

roidata_to_params(*roidata: ndarray*) → DataSetGroup

Convert ROI array data to ROI dataset group.

Parameters
roidata (*numpy.ndarray*) – ROI array data

Returns
ROI dataset group

Return type
DataSetGroup

serialize(*writer: HDF5Writer | JSONWriter | INIWriter*) → None

Serialize the dataset

Parameters
writer (*HDF5Writer | JSONWriter | INIWriter*) – writer object

set_annotations_from_file(*filename: str*) → None

Set object annotations from file (JSON).

Parameters
filename (*str*) – filename

set_defaults() → None

Set default values

classmethod set_global_prop(*realm: str*, ***kwargs*) → None

Set global properties for all data items in the dataset

Parameters

- **realm** (*str*) – realm name
- **kwargs** (*dict*) – properties to set

set_metadata_option(*name: str*, *value: Any*) → None

Set metadata option value

A metadata option is a metadata entry starting with an underscore. It is a way to store application-specific options in object metadata.

Parameters

- **name** (*str*) – option name
- **value** (*Any*) – option value

Valid option names:
'format': format string (str) 'showlabel': show label (bool)

property short_id

Short object ID

text_edit() → None

Edit data set with text input only

to_string(*debug: bool | None = False*, *indent: str | None = None*, *align: bool | None = False*, *show_hidden: bool | None = True*) → str

Return readable string representation of the data set If debug is True, add more details on data items

Parameters

- **debug** (*bool*) – if True, add more details on data items
- **indent** (*str*) – indentation string (default is None, meaning no indentation)
- **align** (*bool*) – if True, align data items (default is False)
- **show_hidden** (*bool*) – if True, show hidden data items (default is True)

Returns
string representation of the data set

Return type
str

transform_shapes(*orig*, *func*, *param=None*)

Apply transform function to result shape / annotations coordinates.

Parameters

- **orig** (*BaseObj*) – original object
- **func** (*callable*) – transform function
- **param** (*object*) – transform function parameter

update_metadata_view_settings() → None

Update metadata view settings from Conf.view

update_resultshapes_from(*other: BaseObj*) → None

Update geometric shape from another object (merge metadata).

Parameters
other – other object, from which to update this object

view(*parent: QWidget | None = None*, *wordwrap: bool = True*, *size: QSize | tuple[int, int] | None = None*) → None

Open a dialog box to view data set

Parameters

- **parent** – parent widget (default is None, meaning no parent)
- **wordwrap** – if True, comment text is wordwrapped
- **size** – dialog size (QSize object or integer tuple (width, height))

write_config(*conf: UserConfig*, *section: str*, *option: str*) → None

Write configuration to a UserConfig instance

Parameters

- **conf** (*UserConfig*) – UserConfig instance
- **section** (*str*) – section name
- **option** (*str*) – option name

cdl.obj.**read_image**(*filename: str*) → *ImageObj*

Read an image from a file.

Parameters
filename (*str*) – File name.

Returns
Image.

Return type
Image

cdl.obj.**create_image**(*title: str*, *data: ndarray | None = None*, *metadata: dict | None = None*, *units: tuple | None = None*, *labels: tuple | None = None*) → *ImageObj*

Create a new Image object

Parameters

- **title** (*str*) – image title
- **data** (*numpy.ndarray*) – image data
- **metadata** (*dict*) – image metadata
- **units** (*tuple*) – X, Y, Z units (tuple of strings)
- **labels** (*tuple*) – X, Y, Z labels (tuple of strings)

Returns
image object

Return type
ImageObj

cdl.obj.**create_image_from_param**(*newparam: NewImageParam*, *addparam: gds.DataSet | None = None*, *edit: bool = False*, *parent: QW.QWidget | None = None*) → *ImageObj* | None

Create a new Image object from dialog box.

Parameters

- **newparam** (*NewImageParam*) – new image parameters
- **addparam** (*guidata.dataset.DataSet*) – additional parameters
- **edit** (*bool*) – Open a dialog box to edit parameters (default: False)
- **parent** (*QWidget*) – parent widget

Returns
new image object or None if user cancelled

Return type
ImageObj

cdl.obj.**new_image_param**(*title: str | None = None*, *itype: ImageTypes | None = None*, *height: int | None = None*, *width: int | None = None*, *dtype: ImageDatatypes | None = None*) → *NewImageParam*

Create a new Image dataset instance.

Parameters

- **title** (*str*) – dataset title (default: None, uses default title)
- **itype** (*ImageTypes*) – image type (default: None, uses default type)

- **height** (*int*) – image height (default: None, uses default height)
- **width** (*int*) – image width (default: None, uses default width)
- **dtype** (ImageDatatypes) – image data type (default: None, uses default data type)

Returns
new image dataset instance

Return type
NewImageParam

class cdl.obj.**ImageTypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Image types

class cdl.obj.**NewImageParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

New image dataset

class cdl.obj.**Gauss2DParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

2D Gaussian parameters

class cdl.obj.**RoiDataGeometries**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ROI data geometry types

class cdl.obj.**ImageDatatypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Image data types

classmethod from_dtype(*dtype*)

Return member from NumPy dtype

classmethod check()

Check if data types are valid

3.4 Computation (cdl.core.computation)

This package contains the computation functions used by the DataLab project. Those functions operate directly on DataLab objects (i.e. *cdl.obj.SignalObj* and *cdl.obj.ImageObj*) and are designed to be used in the DataLab pipeline, but can be used independently as well.

See also:

The *cdl.core.computation* package is the main entry point for the DataLab computation functions when manipulating DataLab objects. See the *cdl.algorithms* package for algorithms that operate directly on NumPy arrays.

Each computation module defines a set of computation objects, that is, functions that implement processing features and classes that implement the corresponding parameters (in the form of `guidata.dataset.datatypes.Dataset` subclasses). The computation functions takes a DataLab object (e.g. *cdl.obj.SignalObj*) and a parameter object (e.g. `cdl.param.MovingAverageParam`) as input and return a DataLab object as output (the result of the computation). The parameter object is used to configure the computation function (e.g. the size of the moving average window).

In DataLab overall architecture, the purpose of this package is to provide the computation functions that are used by the `cdl.core.gui.processor` module, based on the algorithms defined in the *cdl.algorithms* module and on the data model defined in the *cdl.obj* (or `cdl.core.model`) module.

The computation modules are organized in subpackages according to their purpose. The following subpackages are available:

- *cdl.core.computation.base*: Common processing features
- *cdl.core.computation.signal*: Signal processing features
- *cdl.core.computation.image*: Image processing features

3.4.1 Common processing features

class cdl.core.computation.base.**GaussianParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Gaussian filter parameters

class cdl.core.computation.base.**MovingAverageParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Moving average parameters

class cdl.core.computation.base.**MovingMedianParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Moving median parameters

class cdl.core.computation.base.**ThresholdParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Threshold parameters

class cdl.core.computation.base.**ClipParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Data clipping parameters

class cdl.core.computation.base.**ROIDataParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

ROI Editor data

classmethod create(*roidata: ndarray | None = None*, *singleobj: bool | None = None*)

Create ROIDataParam instance

property is_empty: bool

Return True if there is no ROI

class cdl.core.computation.base.**FFTParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

FFT parameters

3.4.2 Signal processing features

cdl.core.computation.signal.**dst_11**(*src:* SignalObj, *name: str*, *suffix: str | None = None*) → *SignalObj*

Create result signal object for compute_11 function

Parameters

- **src** (*SignalObj*) – source signal
- **name** (*str*) – name of the function

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**Wrap11Func**(*func: Callable*)

Wrap a 1 array -> 1 array function (the simple case of y1 = f(y0)) to produce a 1 `SignalObj` -> 1 `SignalObj` function, which can be used inside DataLab's infrastructure to perform computations with `SignalProcessor`.

This wrapping mechanism using a class is necessary for the resulted function to be pickable by the `multiprocessing` module.

The instance of this wrapper is callable and returns a `SignalObj` object.

Example

```
>>> import numpy as np
>>> from cdl.core.computation.signal import Wrap11Func
>>> import cdl.obj
>>> def square(y):
...     return y**2
>>> compute_square = Wrap11Func(square)
>>> x = np.linspace(0, 10, 100)
>>> y = np.sin(x)
>>> sig0 = cdl.obj.create_signal("Example", x, y)
>>> sig1 = compute_square(sig0)
```

Parameters
func – 1 array -> 1 array function

cdl.core.computation.signal.**dst_n1n**(*src1:* SignalObj, *src2:* SignalObj, *name: str*, *suffix: str | None = None*)

Create result signal object for compute_n1n function

Parameters

- **src1** (`SignalObj`) – source signal 1
- **src2** (`SignalObj`) – source signal 2
- **name** (`str`) – name of the function

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_add**(*dst:* SignalObj, *src:* SignalObj) → *SignalObj*

Add signal to result signal :param dst: destination signal :type dst: SignalObj :param src: source signal :type src: SignalObj

cdl.core.computation.signal.**compute_product**(*dst:* SignalObj, *src:* SignalObj) → *SignalObj*

Multiply signal to result signal :param dst: destination signal :type dst: SignalObj :param src: source signal :type src: SignalObj

cdl.core.computation.signal.**compute_difference**(*src1:* SignalObj, *src2:* SignalObj) → *SignalObj*

Compute difference between two signals :param src1: source signal 1 :type src1: SignalObj :param src2: source signal 2 :type src2: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_quadratic_difference**(*src1:* SignalObj, *src2:* SignalObj) → *SignalObj*

Compute quadratic difference between two signals :param src1: source signal 1 :type src1: SignalObj :param src2: source signal 2 :type src2: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_division**(*src1:* SignalObj, *src2:* SignalObj) → *SignalObj*

Compute division between two signals :param src1: source signal 1 :type src1: SignalObj :param src2: source signal 2 :type src2: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**extract_multiple_roi**(*src:* SignalObj, *group:* *DataSetGroup*) → *SignalObj*

Extract multiple regions of interest from data :param src: source signal :type src: SignalObj :param group: group of parameters :type group: gds.DataSetGroup

Returns
signal with multiple regions of interest

Return type
SignalObj

cdl.core.computation.signal.**extract_single_roi**(*src:* SignalObj, *p:* *DataSet*) → *SignalObj*

Extract single region of interest from data :param src: source signal :type src: SignalObj :param p: parameters :type p: gds.DataSet

Returns
signal with single region of interest

Return type
SignalObj

cdl.core.computation.signal.**compute_swap_axes**(*src:* SignalObj) → *SignalObj*

Swap axes :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_abs**(*src:* SignalObj) → *SignalObj*

Compute absolute value :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_re**(*src:* SignalObj) → *SignalObj*

Compute real part :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_im**(*src:* SignalObj) → *SignalObj*

Compute imaginary part :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**DataTypeSParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Convert signal data type parameters

cdl.core.computation.signal.**compute_astype**(*src:* SignalObj, *p:* DataTypeSParam) → *SignalObj*

Convert data type :param src: source signal :param p: parameters

Returns
Result signal object

cdl.core.computation.signal.**compute_log10**(*src:* SignalObj) → *SignalObj*

Compute Log10 :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**PeakDetectionParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Peak detection parameters

cdl.core.computation.signal.**compute_peak_detection**(*src:* SignalObj, *p:* PeakDetectionParam) → *SignalObj*

Peak detection :param src: source signal :type src: SignalObj :param p: parameters :type p: PeakDetectionParam

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**NormalizeYParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Normalize parameters

cdl.core.computation.signal.**compute_normalize**(*src:* SignalObj, *p:* NormalizeYParam) → *SignalObj*

Normalize data :param src: source signal :type src: SignalObj :param p: parameters :type p: NormalizeYParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_derivative**(*src:* SignalObj) → *SignalObj*

Compute derivative :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_integral**(*src:* SignalObj) → *SignalObj*

Compute integral :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**XYCalibrateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Signal calibration parameters

cdl.core.computation.signal.**compute_calibration**(*src:* SignalObj, *p:* XYCalibrateParam) → *SignalObj*

Compute linear calibration :param src: source signal :type src: SignalObj :param p: parameters :type p: XYCalibrateParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_threshold**(*src:* SignalObj, *p:* ThresholdParam) → *SignalObj*

Compute threshold clipping :param src: source signal :type src: SignalObj :param p: parameters :type p: ThresholdParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_clip**(*src:* SignalObj, *p:* ClipParam) → *SignalObj*

Compute maximum data clipping :param src: source signal :type src: SignalObj :param p: parameters :type p: ClipParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_gaussian_filter**(*src:* SignalObj, *p:* GaussianParam) → *SignalObj*

Compute gaussian filter :param src: source signal :type src: SignalObj :param p: parameters :type p: GaussianParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_moving_average**(*src:* SignalObj, *p:* MovingAverageParam) → *SignalObj*

Compute moving average :param src: source signal :type src: SignalObj :param p: parameters :type p: MovingAverageParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_moving_median**(*src:* SignalObj, *p:* MovingMedianParam) → *SignalObj*

Compute moving median :param src: source signal :type src: SignalObj :param p: parameters :type p: MovingMedianParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_wiener**(*src:* SignalObj) → *SignalObj*

Compute Wiener filter :param src: source signal :type src: SignalObj

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_fft**(*src:* SignalObj, *p:* FFTParam) → *SignalObj*

Compute FFT :param src: source signal :type src: SignalObj :param p: parameters :type p: FFTParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_ifft**(*src:* SignalObj, *p:* FFTParam) → *SignalObj*

Compute iFFT :param src: source signal :type src: SignalObj :param p: parameters :type p: FFTParam

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**PolynomialFitParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Polynomial fitting parameters

class cdl.core.computation.signal.**FWHMParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

FWHM parameters

cdl.core.computation.signal.**compute_fwhm**(*signal:* SignalObj, *param:* FWHMParam)

Compute FWHM

cdl.core.computation.signal.**compute_fw1e2**(*signal:* SignalObj)

Compute FW at $1/e^2$

class cdl.core.computation.signal.**InterpolationParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Interpolation parameters

cdl.core.computation.signal.**compute_interpolation**(*src1:* SignalObj, *src2:* SignalObj, *p:* InterpolationParam) → *SignalObj*

Interpolate data :param src1: source signal 1 :type src1: SignalObj :param src2: source signal 2 :type src2: SignalObj :param p: parameters :type p: InterpolationParam

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**ResamplingParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Resample parameters

cdl.core.computation.signal.**compute_resampling**(*src:* SignalObj, *p:* ResamplingParam) → *SignalObj*

Resample data :param src: source signal :type src: SignalObj :param p: parameters :type p: ResampleParam

Returns
result signal object

Return type
SignalObj

class cdl.core.computation.signal.**DetrendingParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Detrending parameters

cdl.core.computation.signal.**compute_detrending**(*src:* SignalObj, *p:* DetrendingParam) → *SignalObj*

Detrend data :param src: source signal :type src: SignalObj :param p: parameters :type p: DetrendingParam

Returns
result signal object

Return type
SignalObj

cdl.core.computation.signal.**compute_convolution**(*src1:* SignalObj, *src2:* SignalObj) → *SignalObj*

Compute convolution of two signals :param src1: source signal 1 :type src1: SignalObj :param src2: source signal 2 :type src2: SignalObj

Returns
result signal object

Return type
SignalObj

3.4.3 Image processing features

Base image processing features

cdl.core.computation.image.**dst_11**(*src:* ImageObj, *name: str*, *suffix: str | None = None*) → *ImageObj*

Create result image object for compute_11 function

Parameters

- **src** (ImageObj) – input image object
- **name** (str) – name of the processing function

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**Wrap11Func**(*func: Callable*)

Wrap a 1 array -> 1 array function to produce a 1 `ImageObj` -> 1 `ImageObj` function, which can be used inside DataLab's infrastructure to perform computations with `ImageProcessor`.

This wrapping mechanism using a class is necessary for the resulted function to be pickable by the `multiprocessing` module.

The instance of this wrapper is callable and returns a `ImageObj` object.

Example

```
>>> import numpy as np
>>> from cdl.core.computation.signal import Wrap11Func
>>> import cdl.obj
>>> def add_noise(data):
...     return data + np.random.random(data.shape)
>>> compute_add_noise = Wrap11Func(add_noise)
>>> data= np.ones((100, 100))
>>> ima0 = cdl.obj.create_image("Example", data)
>>> ima1 = compute_add_noise(ima0)
```

Parameters
func – 1 array -> 1 array function

cdl.core.computation.image.**dst_11_signal**(*src:* ImageObj, *name: str*, *suffix: str | None = None*) → *SignalObj*

Create result signal object for compute_11 function

Parameters

- **src** (ImageObj) – input image object
- **name** (*str*) – name of the processing function

Returns
output signal object

Return type
SignalObj

cdl.core.computation.image.**dst_n1n**(*src1:* ImageObj, *src2:* ImageObj, *name: str*, *suffix: str | None = None*) → *ImageObj*

Create result image object for compute_n1n function

Parameters

- **src1** (ImageObj) – input image object
- **src2** (ImageObj) – input image object
- **name** (*str*) – name of the processing function

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_add**(*dst:* ImageObj, *src:* ImageObj) → *ImageObj*

Compute addition between two images

Parameters

- **dst** (ImageObj) – output image object
- **src** (ImageObj) – input image object

cdl.core.computation.image.**compute_product**(*dst:* ImageObj, *src:* ImageObj) → *ImageObj*

Compute product between two images

Parameters

- **dst** (ImageObj) – output image object
- **src** (ImageObj) – input image object

cdl.core.computation.image.**compute_difference**(*src1:* ImageObj, *src2:* ImageObj) → *ImageObj*

Compute difference between two images :param src1: input image object :type src1: ImageObj :param src2: input image object :type src2: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_quadratic_difference**(*src1:* ImageObj, *src2:* ImageObj) → *ImageObj*

Compute quadratic difference between two images :param src1: input image object :type src1: ImageObj :param src2: input image object :type src2: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_division**(*src1:* ImageObj, *src2:* ImageObj) → *ImageObj*

Compute division between two images :param src1: input image object :type src1: ImageObj :param src2: input image object :type src2: ImageObj

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**FlatFieldParam**(*title=None*, *comment=None*, *icon=''*)

Flat-field parameters

cdl.core.computation.image.**compute_flatfield**(*src1:* ImageObj, *src2:* ImageObj, *p:* FlatFieldParam) → *ImageObj*

Compute flat field correction :param src1: raw data image object :type src1: ImageObj :param src2: flat field image object :type src2: ImageObj :param p: flat field parameters :type p: FlatFieldParam

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**LogP1Param**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Log10 parameters

cdl.core.computation.image.**compute_logp1**(*src:* ImageObj, *p:* LogP1Param) → *ImageObj*

Compute log10(z+n) :param src: input image object :type src: ImageObj :param p: parameters :type p: LogP1Param

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**RotateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Rotate parameters

cdl.core.computation.image.**rotate_obj_coords**(*angle: float*, *obj:* ImageObj, *orig:* ImageObj, *coords: ndarray*) → None

Apply rotation to coords associated to image obj :param angle: rotation angle (in degrees) :type angle: float :param obj: image object :type obj: ImageObj :param orig: original image object :type orig: ImageObj :param coords: coordinates to rotate :type coords: numpy.ndarray

Returns
output data

Return type
np.ndarray

cdl.core.computation.image.**rotate_obj_alpha**(*obj:* ImageObj, *orig:* ImageObj, *coords:* *ndarray*, *p:* RotateParam) → None

Apply rotation to coords associated to image obj

cdl.core.computation.image.**compute_rotate**(*src:* ImageObj, *p:* RotateParam) → *ImageObj*

Rotate data :param src: input image object :type src: ImageObj :param p: parameters :type p: RotateParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**rotate_obj_90**(*dst:* ImageObj, *src:* ImageObj, *coords:* *ndarray*) → None

Apply rotation to coords associated to image obj

cdl.core.computation.image.**compute_rotate90**(*src:* ImageObj) → *ImageObj*

Rotate data 90° :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**rotate_obj_270**(*dst:* ImageObj, *src:* ImageObj, *coords:* *ndarray*) → None

Apply rotation to coords associated to image obj

cdl.core.computation.image.**compute_rotate270**(*src:* ImageObj) → *ImageObj*

Rotate data 270° :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**hflip_coords**(*dst:* ImageObj, *src:* ImageObj, *coords:* *ndarray*) → None

Apply HFlip to coords

cdl.core.computation.image.**compute_fliph**(*src:* ImageObj) → *ImageObj*

Flip data horizontally :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**vflip_coords**(*dst:* ImageObj, *src:* ImageObj, *coords:* *ndarray*) → None

Apply VFlip to coords

cdl.core.computation.image.**compute_flipv**(*src:* ImageObj) → *ImageObj*

Flip data vertically :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**GridParam**(*title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False*)

Grid parameters

class cdl.core.computation.image.**ResizeParam**(*title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False*)

Resize parameters

cdl.core.computation.image.**compute_resize**(*src:* ImageObj, *p:* ResizeParam) → *ImageObj*

Zooming function :param src: input image object :type src: ImageObj :param p: parameters :type p: ResizeParam

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**BinningParam**(*title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False*)

Binning parameters

cdl.core.computation.image.**compute_binning**(*src:* ImageObj, *param:* BinningParam) → *ImageObj*

Binning function on data :param src: input image object :type src: ImageObj :param param: parameters :type param: BinningParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**extract_multiple_roi**(*src:* ImageObj, *group: DataSetGroup*) → *ImageObj*

Extract multiple regions of interest from data :param src: input image object :type src: ImageObj :param group: parameters defining the regions of interest :type group: gds.DataSetGroup

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**extract_single_roi**(*src:* ImageObj, *p: DataSet*) → *ImageObj*

Extract single ROI :param src: input image object :type src: ImageObj :param p: ROI parameters :type p: gds.DataSet

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**ProfileParam**(*title: str | None = None, comment: str | None = None, icon: str = '', readonly: bool = False*)

Horizontal or vertical profile parameters

cdl.core.computation.image.**compute_profile**(*src:* ImageObj, *p:* ProfileParam) → *ImageObj*

Compute horizontal or vertical profile :param src: input image object :param p: parameters

Returns
Output image object

class cdl.core.computation.image.**AverageProfileParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Average horizontal or vertical profile parameters

cdl.core.computation.image.**compute_average_profile**(*src:* ImageObj, *p:* AverageProfileParam) → *ImageObj*

Compute horizontal or vertical average profile :param src: input image object :param p: parameters

Returns
Output image object

class cdl.core.computation.image.**RadialProfileParam**(**args*, ***kwargs*)

Radial profile parameters

update_from_image(*obj:* ImageObj) → None

Update parameters from image

choice_callback(*item*, *value*)

Callback for choice item

cdl.core.computation.image.**compute_radial_profile**(*src:* ImageObj, *p:* RadialProfileParam) → *ImageObj*

Compute radial profile around the centroid

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.**compute_swap_axes**(*src:* ImageObj) → *ImageObj*

Swap image axes :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_abs**(*src:* ImageObj) → *ImageObj*

Compute absolute value :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_re**(*src:* ImageObj) → *ImageObj*

Compute real part :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_im**(*src:* ImageObj) → *ImageObj*

Compute imaginary part :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**DataTypeIParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Convert image data type parameters

cdl.core.computation.image.**compute_astype**(*src:* ImageObj, *p:* DataTypeIParam) → *ImageObj*

Convert image data type :param src: input image object :param p: parameters

Returns
Output image object

cdl.core.computation.image.**compute_log10**(*src:* ImageObj) → *ImageObj*

Compute log10 :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**ZCalibrateParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Image linear calibration parameters

cdl.core.computation.image.**compute_calibration**(*src:* ImageObj, *p:* ZCalibrateParam) → *ImageObj*

Compute linear calibration :param src: input image object :type src: ImageObj :param param: calibration parameters :type param: ZCalibrateParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_threshold**(*src:* ImageObj, *p:* ThresholdParam) → *ImageObj*

Apply thresholding :param src: input image object :type src: ImageObj :param p: parameters :type p: ThresholdParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_clip**(*src:* ImageObj, *p:* ClipParam) → *ImageObj*

Apply clipping :param src: input image object :type src: ImageObj :param p: parameters :type p: ClipParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_gaussian_filter**(*src:* ImageObj, *p:* GaussianParam) → *ImageObj*

Compute gaussian filter :param src: input image object :type src: ImageObj :param p: parameters :type p: GaussianParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_moving_average**(*src:* ImageObj, *p:* MovingAverageParam) → *ImageObj*

Compute moving average :param src: input image object :type src: ImageObj :param p: parameters :type p: MovingAverageParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_moving_median**(*src:* ImageObj, *p:* MovingMedianParam) → *ImageObj*

Compute moving median :param src: input image object :type src: ImageObj :param p: parameters :type p: MovingMedianParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_wiener**(*src:* ImageObj) → *ImageObj*

Compute Wiener filter :param src: input image object :type src: ImageObj

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_fft**(*src:* ImageObj, *p:* FFTParam) → *ImageObj*

Compute FFT :param src: input image object :type src: ImageObj :param p: parameters :type p: FFTParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**compute_ifft**(*src:* ImageObj, *p:* FFTParam) → *ImageObj*

Compute inverse FFT :param src: input image object :type src: ImageObj :param p: parameters :type p: FFTParam

Returns
output image object

Return type
ImageObj

class cdl.core.computation.image.**ButterworthParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Butterworth filter parameters

cdl.core.computation.image.**compute_butterworth**(*src:* ImageObj, *p:* ButterworthParam) → *ImageObj*

Compute Butterworth filter :param src: input image object :type src: ImageObj :param p: parameters :type p: ButterworthParam

Returns
output image object

Return type
ImageObj

cdl.core.computation.image.**calc_with_osr**(*image:* ImageObj, *func: Callable*, **args: Any*) → ndarray

Exec computation taking into account image x0, y0, dx, dy and ROIs

Parameters

- **image** – input image object
- **func** – computation function
- ***args** – computation function arguments

Returns
Computation result

> **Warning:** The computation function must take either a single argument (the data) or multiple arguments (the data followed by the computation parameters).
>
> Moreover, the computation function must return a single value or a NumPy array containing the result of the computation. This array contains the coordinates of points, polygons, circles or ellipses in the form [[x, y], ...], or [[x0, y0, x1, y1, ...], ...], or [[x0, y0, r], ...], or [[x0, y0, a, b, theta], ...].

cdl.core.computation.image.**get_centroid_coords**(*data: ndarray*) → ndarray

Return centroid coordinates :param data: input data :type data: numpy.ndarray

Returns
centroid coordinates

Return type
np.ndarray

cdl.core.computation.image.**compute_centroid**(*image:* ImageObj) → ndarray

Compute centroid :param image: input image :type image: ImageObj

Returns
centroid coordinates

Return type
np.ndarray

cdl.core.computation.image.**get_enclosing_circle_coords**(*data: ndarray*) → ndarray

Return diameter coords for the circle contour enclosing image values above threshold (FWHM) :param data: input data :type data: numpy.ndarray

Returns
diameter coords

Return type
np.ndarray

cdl.core.computation.image.**compute_enclosing_circle**(*image:* ImageObj) → ndarray

Compute minimum enclosing circle :param image: input image :type image: ImageObj

Returns
diameter coords

Return type
np.ndarray

class cdl.core.computation.image.**HoughCircleParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Circle Hough transform parameters

cdl.core.computation.image.**compute_hough_circle_peaks**(*image:* ImageObj, *p:* HoughCircleParam) → ndarray

Compute Hough circles :param image: input image :type image: ImageObj :param p: parameters :type p: HoughCircleParam

Returns
circle coordinates

Return type
np.ndarray

Exposure correction features

Exposure computation module

class cdl.core.computation.image.exposure.**AdjustGammaParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Gamma adjustment parameters

cdl.core.computation.image.exposure.**compute_adjust_gamma**(*src:* ImageObj, *p:* AdjustGammaParam) → *ImageObj*

Gamma correction

Parameters

- **src** – input image object
- **p** (AdjustGammaParam) – parameters

Returns
Output image object

class cdl.core.computation.image.exposure.**AdjustLogParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Logarithmic adjustment parameters

cdl.core.computation.image.exposure.**compute_adjust_log**(*src:* ImageObj, *p:* AdjustLogParam) → *ImageObj*

Compute log correction

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.exposure.**AdjustSigmoidParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Sigmoid adjustment parameters

cdl.core.computation.image.exposure.**compute_adjust_sigmoid**(*src:* ImageObj, *p:* AdjustSigmoidParam) → *ImageObj*

Compute sigmoid correction

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.exposure.**RescaleIntensityParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Intensity rescaling parameters

cdl.core.computation.image.exposure.**compute_rescale_intensity**(*src:* ImageObj, *p:* RescaleIntensityParam) → *ImageObj*

Rescale image intensity levels

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.exposure.**EqualizeHistParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Histogram equalization parameters

cdl.core.computation.image.exposure.**compute_equalize_hist**(*src:* ImageObj, *p:* EqualizeHistParam) → *ImageObj*

Histogram equalization

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.exposure.**EqualizeAdaptHistParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Adaptive histogram equalization parameters

cdl.core.computation.image.exposure.**compute_equalize_adapthist**(*src:* ImageObj, *p:* EqualizeAdaptHistParam) → *ImageObj*

Adaptive histogram equalization

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

Restoration features

Restoration computation module

class cdl.core.computation.image.restoration.**DenoiseTVParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Total Variation denoising parameters

cdl.core.computation.image.restoration.**compute_denoise_tv**(*src:* ImageObj, *p:* DenoiseTVParam) → *ImageObj*

Compute Total Variation denoising

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.restoration.**DenoiseBilateralParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Bilateral filter denoising parameters

cdl.core.computation.image.restoration.**compute_denoise_bilateral**(*src:* ImageObj, *p:* DenoiseBilateralParam) → *ImageObj*

Compute bilateral filter denoising

Parameters

- **src** – input image object

- **p** – parameters

Returns
Output image object

class cdl.core.computation.image.restoration.**DenoiseWaveletParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Wavelet denoising parameters

cdl.core.computation.image.restoration.**compute_denoise_wavelet**(*src:* ImageObj, *p:* DenoiseWaveletParam) → *ImageObj*

Compute Wavelet denoising

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.restoration.**compute_denoise_tophat**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Denoise using White Top-Hat

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

Morphological features

Morphology computation module

class cdl.core.computation.image.morphology.**MorphologyParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

White Top-Hat parameters

cdl.core.computation.image.morphology.**compute_white_tophat**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute White Top-Hat

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.morphology.**compute_black_tophat**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute Black Top-Hat

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.morphology.**compute_erosion**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute Erosion

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.morphology.**compute_dilation**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute Dilation

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.morphology.**compute_opening**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute morphological opening

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.morphology.**compute_closing**(*src:* ImageObj, *p:* MorphologyParam) → *ImageObj*

Compute morphological closing

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

Edge detection features

Edges computation module

class cdl.core.computation.image.edges.**CannyParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Canny filter parameters

cdl.core.computation.image.edges.**compute_canny**(*src:* ImageObj, *p:* CannyParam) → *ImageObj*

Compute Canny filter

Parameters

- **src** – input image object
- **p** – parameters

Returns
Output image object

cdl.core.computation.image.edges.**compute_roberts**(*src:* ImageObj) → *ImageObj*

Compute Roberts filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_prewitt**(*src:* ImageObj) → *ImageObj*

Compute Prewitt filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_prewitt_h**(*src:* ImageObj) → *ImageObj*

Compute horizontal Prewitt filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_prewitt_v**(*src:* ImageObj) → *ImageObj*

Compute vertical Prewitt filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_sobel**(*src:* ImageObj) → *ImageObj*

Compute Sobel filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_sobel_h**(*src:* ImageObj) → *ImageObj*

Compute horizontal Sobel filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_sobel_v**(*src:* ImageObj) → *ImageObj*

Compute vertical Sobel filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_scharr**(*src:* ImageObj) → *ImageObj*

Compute Scharr filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_scharr_h**(*src:* ImageObj) → *ImageObj*

Compute horizontal Scharr filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_scharr_v**(*src:* ImageObj) → *ImageObj*

Compute vertical Scharr filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_farid**(*src:* ImageObj) → *ImageObj*

Compute Farid filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_farid_h**(*src:* ImageObj) → *ImageObj*

Compute horizontal Farid filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_farid_v**(*src:* ImageObj) → *ImageObj*

Compute vertical Farid filter

Parameters
src – input image object

Returns
Output image object

cdl.core.computation.image.edges.**compute_laplace**(*src:* ImageObj) → *ImageObj*

Compute Laplace filter

Parameters
src – input image object

Returns
Output image object

Detection features

Blob detection computation module

class cdl.core.computation.image.detection.**GenericDetectionParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Generic detection parameters

class cdl.core.computation.image.detection.**Peak2DDetectionParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Peak detection parameters

cdl.core.computation.image.detection.**compute_peak_detection**(*image:* ImageObj, *p:* Peak2DDetectionParam) → ndarray

Compute 2D peak detection

Parameters
- **imageOutput** – input image
- **p** – parameters

Returns
Peak coordinates

class cdl.core.computation.image.detection.**ContourShapeParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Contour shape parameters

cdl.core.computation.image.detection.**compute_contour_shape**(*image:* ImageObj, *p:* ContourShapeParam) → ndarray

Compute contour shape fit

class cdl.core.computation.image.detection.**BaseBlobParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Base class for blob detection parameters

class cdl.core.computation.image.detection.**BlobDOGParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Difference of Gaussian method

cdl.core.computation.image.detection.**compute_blob_dog**(*image:* ImageObj, *p:* BlobDOGParam) → ndarray

Compute blobs using Difference of Gaussian method

Parameters

- **imageOutput** – input image
- **p** – parameters

Returns

Blobs coordinates

class cdl.core.computation.image.detection.**BlobDOHParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Determinant of Hessian method

cdl.core.computation.image.detection.**compute_blob_doh**(*image:* ImageObj, *p:* BlobDOHParam) → ndarray

Compute blobs using Determinant of Hessian method

Parameters

- **imageOutput** – input image
- **p** – parameters

Returns

Blobs coordinates

class cdl.core.computation.image.detection.**BlobLOGParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using Laplacian of Gaussian method

cdl.core.computation.image.detection.**compute_blob_log**(*image:* ImageObj, *p:* BlobLOGParam) → ndarray

Compute blobs using Laplacian of Gaussian method

Parameters

- **imageOutput** – input image
- **p** – parameters

Returns

Blobs coordinates

class cdl.core.computation.image.detection.**BlobOpenCVParam**(*title: str | None = None*, *comment: str | None = None*, *icon: str = ''*, *readonly: bool = False*)

Blob detection using OpenCV

cdl.core.computation.image.detection.**compute_blob_opencv**(*image:* ImageObj, *p:* BlobOpenCVParam) → ndarray

Compute blobs using OpenCV

Parameters

- **imageOutput** – input image
- **p** – parameters

Returns

Blobs coordinates

3.5 Proxy objects (cdl.proxy)

The *cdl.proxy* module provides a way to access DataLab features from a proxy class.

3.5.1 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 cdl.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 (*bool*) – 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 cdl.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
```

(continues on next page)

(continued from previous page)

```
>>> proxy.get_object_titles()
['toto']
>>> proxy["toto"]  # from title
<cdl.core.model.signal.SignalObj at 0x7f7f1c0b4a90>
>>> proxy[1]  # from number
<cdl.core.model.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** (*list*) – annotation plot items
- **refresh_plot** (*bool | None*) – refresh plot. Defaults to True.
- **panel** (*str | None*) – panel name (valid values: “signal”, “image”). If None, current panel is used.

add_image(*title: str*, *data: ndarray*, *xunit: str | None = None*, *yunit: str | None = None*, *zunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*, *zlabel: str | None = None*) → bool

Add image data to DataLab.

Parameters

- **title** (*str*) – Image title
- **data** (*numpy.ndarray*) – Image data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.
- **zunit** (*str | None*) – Z unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.
- **zlabel** (*str | None*) – Z label. Defaults to None.

Returns
True if image was added successfully, False otherwise

Return type
bool

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** (*str | None*) – Label title. Defaults to None. If None, the title is the object title.
- **panel** (*str | None*) – panel name (valid values: “signal”, “image”). If None, current panel is used.

add_object(*obj:* SignalObj | ImageObj) → None

Add object to DataLab.

Parameters
obj (SignalObj | ImageObj) – Signal or image object

add_signal(*title: str*, *xdata: ndarray*, *ydata: ndarray*, *xunit: str | None = None*, *yunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*) → bool

Add signal data to DataLab.

Parameters

- **title** (*str*) – Signal title
- **xdata** (*numpy.ndarray*) – X data
- **ydata** (*numpy.ndarray*) – Y data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.

Returns
True if signal was added successfully, False otherwise

Return type
bool

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

calc(*name: str*, *param: DataSet | None = None*) → DataSet

Call compute function `name` in current panel's processor.

Parameters

- **name** (*str*) – Compute function name
- **param** (*guidata.dataset.DataSet | None*) – Compute function parameter. Defaults to None.

Returns
Compute function result

Return type
guidata.dataset.DataSet

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** (*str | None*) – XML-RPC port to connect to. If not specified, the port is automatically retrieved from DataLab configuration.
- **timeout** (*float | None*) – Timeout in seconds. Defaults to 5.0.

- **retries** (*int | None*) – Number of retries. Defaults to 10.

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() → Callable

Return a context manager to temporarily disable auto refresh.

Returns
Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.compute_fft()
...     proxy.compute_wiener()
...     proxy.compute_ifft()
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True*, *keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

disconnect() → None

Disconnect from DataLab XML-RPC server.

get_current_panel() → str

Return current panel name.

Returns
Panel name (valid values: “signal”, “image”, “macro”))

Return type
str

get_group_titles_with_object_infos() → 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 (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

Returns
list of object titles

Return type
list[str]

Raises
ValueError – if panel not found

get_object_uuids(*panel: 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 (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

Returns
list of object uuids

Return type
list[str]

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

Return type
list[str]

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 version.

Returns
DataLab version

Return type
str

import_h5_file(*filename: str*, *reset_all: bool | None = None*) → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- **filename** (*str*) – HDF5 file name
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

is_connected() → bool

Return True if connected to DataLab XML-RPC server.

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[str] | None*) – List of HDF5 files to open. Defaults to None.
- **import_all** (*bool | None*) – Import all objects from HDF5 files. Defaults to None.
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

open_object(*filename: str*) → None

Open object from file in current panel (signal/image).

Parameters
filename (*str*) – File name

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters

filename (*str*) – 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** (*str | None*) – 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 (*str*) – Panel name (valid values: "signal", "image", "macro"))

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters

state (*bool*) – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters

state (*bool*) – Show titles state

3.5.2 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 cdl.proxy.**LocalProxy**(*cdl: CDLMainWindow | ServerProxy | None = None*)

DataLab local proxy class.

This class provides access to DataLab features from a proxy class. This is the local version of proxy, which is used when DataLab is started from the same process as the proxy.

Parameters

cdl (*CDLMainWindow*) – CDLMainWindow instance.

add_signal(*title: str*, *xdata: ndarray*, *ydata: ndarray*, *xunit: str | None = None*, *yunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*) → bool

Add signal data to DataLab.

Parameters

- **title** (*str*) – Signal title
- **xdata** (*numpy.ndarray*) – X data
- **ydata** (*numpy.ndarray*) – Y data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.

Returns
True if signal was added successfully, False otherwise

Return type
bool

Raises

- **ValueError** – Invalid xdata dtype
- **ValueError** – Invalid ydata dtype

add_image(*title: str*, *data: ndarray*, *xunit: str | None = None*, *yunit: str | None = None*, *zunit: str | None = None*, *xlabel: str | None = None*, *ylabel: str | None = None*, *zlabel: str | None = None*) → bool

Add image data to DataLab.

Parameters

- **title** (*str*) – Image title
- **data** (*numpy.ndarray*) – Image data
- **xunit** (*str | None*) – X unit. Defaults to None.
- **yunit** (*str | None*) – Y unit. Defaults to None.
- **zunit** (*str | None*) – Z unit. Defaults to None.
- **xlabel** (*str | None*) – X label. Defaults to None.
- **ylabel** (*str | None*) – Y label. Defaults to None.
- **zlabel** (*str | None*) – Z label. Defaults to None.

Returns
True if image was added successfully, False otherwise

Return type
bool

Raises
ValueError – Invalid data dtype

calc(*name: str*, *param: DataSet | None = None*) → DataSet

Call compute function `name` in current panel's processor.

Parameters

- **name** (*str*) – Compute function name
- **param** (*guidata.dataset.DataSet | None*) – Compute function
- **None.** (*parameter. Defaults to*) –

Returns
Compute function result

Return type
guidata.dataset.DataSet

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** (*list*) – annotation plot items
- **refresh_plot** (*bool | None*) – refresh plot. Defaults to True.
- **panel** (*str | None*) – panel name (valid values: “signal”, “image”). If None, current panel is used.

add_object(*obj:* SignalObj | ImageObj) → None

Add object to DataLab.

Parameters
obj (SignalObj | ImageObj) – Signal or image object

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** (*str | None*) – Label title. Defaults to None. If None, the title is the object title.

- **panel** (*str | None*) – panel name (valid values: “signal”, “image”). If None, current panel is used.

close_application() → None

Close DataLab application

context_no_refresh() → Callable

Return a context manager to temporarily disable auto refresh.

Returns
Context manager

Example

```
>>> with proxy.context_no_refresh():
...     proxy.add_image("image1", data1)
...     proxy.compute_fft()
...     proxy.compute_wiener()
...     proxy.compute_ifft()
...     # Auto refresh is disabled during the above operations
```

delete_metadata(*refresh_plot: bool = True*, *keep_roi: bool = False*) → None

Delete metadata of selected objects

Parameters

- **refresh_plot** – Refresh plot. Defaults to True.
- **keep_roi** – Keep ROI. Defaults to False.

get_current_panel() → str

Return current panel name.

Returns
Panel name (valid values: “signal”, “image”, “macro”))

Return type
str

get_group_titles_with_object_infos() → 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 (*str | None*) – panel name (valid values: “signal”, “image”). If None, current panel is used.

Returns
list of object titles

Return type
list[str]

Raises
ValueError – if panel not found

get_object_uuids(*panel: 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 (*str | None*) – panel name (valid values: "signal", "image"). If None, current panel is used.

Returns
list of object uuids

Return type
list[str]

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

Return type
list[str]

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 version.

Returns
DataLab version

Return type
str

import_h5_file(*filename: str*, *reset_all: bool | None = None*) → None

Open DataLab HDF5 browser to Import HDF5 file.

Parameters

- **filename** (*str*) – HDF5 file name
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

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[str] | None*) – List of HDF5 files to open. Defaults to None.
- **import_all** (*bool | None*) – Import all objects from HDF5 files. Defaults to None.
- **reset_all** (*bool | None*) – Reset all application data. Defaults to None.

open_object(*filename: str*) → None

Open object from file in current panel (signal/image).

Parameters

filename (*str*) – File name

raise_window() → None

Raise DataLab window

reset_all() → None

Reset all application data

save_to_h5_file(*filename: str*) → None

Save to a DataLab HDF5 file.

Parameters

filename (*str*) – 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** (*str | None*) – 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 (*str*) – Panel name (valid values: “signal”, “image”, “macro”))

toggle_auto_refresh(*state: bool*) → None

Toggle auto refresh state.

Parameters

state (*bool*) – Auto refresh state

toggle_show_titles(*state: bool*) → None

Toggle show titles state.

Parameters

state (*bool*) – Show titles state

3.5.3 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.

---

cdl.proxy.**proxy_context**(*what: str*) → Generator[*LocalProxy* | *RemoteProxy*, None, None]

Context manager handling CDL proxy creation and destruction.

Parameters

what (*str*) – proxy type (“local” or “remote”) For remote proxy, the port can be specified as “remote:port”

Yields

Generator[LocalProxy | RemoteProxy, None, None] –

proxy

LocalProxy if what == “local” RemoteProxy if what == “remote” or “remote:port”

Example

with proxy_context(“local”) as proxy:
proxy.add_signal(...)

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

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.

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 79 characters.
- Respect the naming conventions (classes, functions, variables, etc.).
- Use specific exceptions instead of the generic `Exception`.

To enforce these guidelines, the following tools are mandatory:

- black for code formatting.
- isort for import sorting.
- pylint for static code analysis.

black

If you are using Visual Studio Code, the project settings will automatically format your code on save.

Or you may use *black* manually. To format your code, run the following command:

```
black .
```

isort

Again, if you are using Visual Studio Code, the project settings will automatically sort your imports on save.

Or you may use *isort* manually. To sort your imports, run the following command:

```
isort .
```

pylint

To run *pylint*, run the following command:

```
pylint datalab
```

If you are using Visual Studio Code on Windows, you may run the task "Run Pylint" to run *pylint* on the project.

Note: A *pylint* rating greater than 9/10 is required to merge a pull request.

Specific coding guidelines

In addition to the generic coding guidelines, we have the following specific guidelines:

- Write docstrings for all classes, methods and functions. The docstrings should follow the Google style.
- Add typing annotations for all functions and methods. The annotations should use the future syntax (`from __future__ import annotations`)
- Try to keep the code as simple as possible. If you have to write a complex piece of code, try to split it into several functions or classes.
- Add as many comments as possible. The code should be self-explanatory, but it is always useful to add some comments to explain the general idea of the code, or to explain some tricky parts.
- Do not use `from module import *` statements, even in the `__init__` module of a package.
- Avoid using mixins (multiple inheritance) when possible. It is often possible to use composition instead of inheritance.
- Avoid using `__getattr__` and `__setattr__` methods. They are often used to implement lazy initialization, but this can be done in a more explicit way.

4.4.3 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
Black Formatter	Python code formatter
gettext	Gettext syntax highlighting
isort	Python import sorter
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

Python Environment

DataLab requires the following :

- Python (e.g. WinPython)
- Additional Python packages

Installing all required packages :

```
pip install --upgrade -r dev\requirements.txt
```

See Installation for more details on reference Python and Qt versions.

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
3.8	OK	3.8.10.0
3.9	OK	3.9.10.0
3.10	OK	3.10.11.1
3.11	OK	3.11.5.0
3.12	OK	3.12.0.1

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 "cdl\data\tests" (public data).

A second folder %CDL_DATA% (optional) may be defined for additional tests which are still under development (or for confidential data).

Specific environment variables

Enable the "debug" mode (no stdin/stdout redirection towards internal console):

```
@REM Mode DEBUG
set DEBUG=1
```

Building PDF documentation requires LaTeX. On Windows, the following environment:

```
@REM LaTeX executable must be in Windows PATH, for mathematical equations rendering
@REM Example with MiKTeX :
set PATH=C:\\Apps\\miktex-portable\\texmfs\\install\\miktex\\bin\\x64;%PATH%
```

Visual Studio Code configuration used in `launch.json` and `tasks.json` (examples) :

```
@REM Development environment
set CDL_PYTHONEXE=C:\C2OIQ-DevCDL\python-3.8.10.amd64\python.exe
@REM Folder containing additional working test data
set CDL_DATA=C:\Dev\Projets\CDL_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).

Third-party Software

The following software may be required for maintaining the project:

Software	Description
gettext	Translations
Git	Version control system
ImageMagick	Image manipulation utilities
Inkscape	Vector graphics editor
MikTeX	LaTeX distribution on Windows

4.4.4 Roadmap

Future milestones

Features

- Add a Jupyter kernel interface to DataLab:
  - This would allow to use DataLab from other software, such as Jupyter notebooks, Spyder or Visual Studio Code
  - This would also allow to share data between DataLab and other software (e.g. display DataLab numerical results in Jupyter notebooks or the other way around, display Jupyter results in DataLab, etc.)
- Create a Jupyter plugin for interactive data analysis with DataLab:
  - Using DataLab from a Jupyter notebook is already possible, thanks to the remote control features (see *Remote controlling*), but it would be nice to have a dedicated plugin
  - This plugin would allow to use DataLab as a Jupyter kernel, and to display DataLab numerical results in Jupyter notebooks or the other way around (e.g. display Jupyter results in DataLab)
  - This plugin would also allow to use DataLab's processing features from Jupyter notebooks
  - A typical use case could also consist in using DataLab for manipulating signals or images efficiently, and using Jupyter for custom data analysis based on specific / home-made algorithms
  - This plugin could be implemented by using the Jupyter kernel interface (see above)
- Create a Spyder plugin for interactive data analysis connected with DataLab:

  - This is exactly the same use case as for the Jupyter plugin, but for Spyder
  - This plugin could also be implemented by using the Jupyter kernel interface (see above)
- Add support for time series (see Issue #27)

Maintenance

- 2024: switch to gRPC for remote control (instead of XML-RPC), if there is a need for a more efficient communication protocol (see Issue #18)
- 2025: drop PyQt5 support (end-of-life: mid-2025), and switch to PyQt6 ; this should be straightforward, thanks to the *qtpy* compatibility layer and to the fact that *PlotPyStack* is already compatible with PyQt6)

Other tasks

- Create a DataLab plugin template (see Issue #26)
- Make tutorial videos: plugin system, remote control features, etc. (see Issue #25)

Past milestones

DataLab 0.11

- Add a drag-and-drop feature to the signal and image panels, to allow reordering signals and images (see Issue #17)
- Add "Move up" and "Move down" buttons to the signal and image panels, to allow reordering signals and images (see Issue #22)
- Add 1D convolution, interpolation, resampling and detrending features

DataLab 0.10

- Develop a very simple DataLab plugin to demonstrate the plugin system
- Serialize curve and image styles in HDF5 files
- Add an "Auto-refresh" global option, to be able to disable the automatic refresh of the main window when doing multiple processing steps, thus improving performance
- Improve curve readability (e.g. avoid dashed lines, use contrasted colors, and use anti-aliasing)

DataLab 0.9

- Python 3.11 is the new reference
- Run computations in a separate process:
  - Execute a "computing server" in background, in another process
  - For each computation, send serialized data and computing function to the server and wait for the result
  - It is then possible to stop any computation at any time by killing the server process and restarting it (eventually after incrementing the communication port number)

- Optimize image displaying performance
- Add preferences dialog box
- Add new image processing features: denoising, ...
- Image processing results: added support for polygon shapes (e.g. for contour detection)
- New plugin system: API for third-party extensions
  - Objective #1: a plugin must be manageable using a single Python script, which includes an extension of *ImageProcessor*, *ActionHandler* and new file format support
  - Objective #2: plugins must be simply stored in a folder wich defaults to the user directory (same folder as ".DataLab.ini" configuration file)
- Add a macro-command system:
  - New embedded Python editor
  - Scripts using the same API as high-level applicative test scenarios
  - Support for macro recording
- Add an xmlrpc server to allow DataLab remote control:
  - Controlling DataLab main features (open a signal or an image, open a HDF5 file, etc.) and processing features (run a computation, etc.)
  - Take control of DataLab from a third-party software
  - Run interactive calculations from an IDE (e.g. Spyder or Visual Studio Code)

4.4.5 Changelog

See DataLab roadmap page for future and past milestones.

DataLab Version 0.12.0

Clarity-Enhanced Interface Update:

- The tabs used to switch between the data panels (signals and images) and the visualization components ("Curve panel" and "Image panel") have been renamed to "Signal Panel" and "Image Panel" (instead of "Signals" and "Images")
- The visualization components have been renamed to "Signal View" and "Image View" (instead of "Curve panel" and "Image panel")
- The data panel toolbar has been renamed to "Signal Toolbar" and "Image Toolbar" (instead of "Signal Processing Toolbar" and "Image Processing Toolbar")
- Ergonomics improvements: the "Signal Panel" and "Image Panel" are now displayed on the left side of the main window, and the "Signal View" and "Image View" are displayed on the right side of the main window. This reduces the distance between the list of objects (signals and images) and the associated actions (toolbars and menus), and makes the interface more intuitive and easier to use

New tour and demo feature:

- When starting DataLab for the first time, an optional tour is now shown to the user to introduce the main features of the application
- The tour can be started again at any time from the "?" menu

- Also added a new "Demo" feature to the "?" menu

New Binder environment to test DataLab online without installing anything

Documentation:

- New text tutorials are available:
  - Measuring Laser Beam Size
  - DataLab and Spyder: a perfect match
- "Getting started" section: added more explanations and links to the tutorials
- New "Contributing" section explaining how to contribute to DataLab, whether you are a developer or not
- New "Macros" section explaining how to use the macro commands feature
- Added "Copy" button to code blocks in the documentation

New features:

- New "Text file import assistant" feature:
  - This feature allows to import text files as signals or images
  - The user can define the source (clipboard or texte file)
  - Then, it is possible to define the delimiter, the number of rows to skip, the destination data type, etc.
- Added menu on the "Signal Panel" and "Image Panel" tabs corner to quickly access the most used features (e.g. "Add", "Remove", "Duplicate", etc.)
- Intensity profile extraction feature:
  - Added graphical user interface to extract intensity profiles from images, for both line and averaged profiles
  - Parameters are still directly editable by the user ("Edit profile parameters" button)
  - Parameters are now stored from one profile extraction to another
- Statistics feature:
  - Added `<y>/(y)` to the signal "Statistics" result table (in addition to the mean, median, standard deviation, etc.)
  - Added `peak-to-peak` to the signal and image "Statistics" result table
- Curve fitting feature: fit results are now stored in a dictionary in the signal metadata (instead of being stored individually in the signal metadata)
- Window state:
  - The toolbars and dock widgets state (visibility, position, etc.) are now stored in the configuration file and restored at startup (size and position were already stored and restored)
  - This implements part of Issue #30 - Save/restore main window layout

Bug fixes:

- Fixed Issue #41 - Radial profile extraction: unable to enter user-defined center coordinates
- Fixed Issue #49 - Error when trying to open a (UTF-8 BOM) text file as an image
- Fixed Issue #51 - Unexpected dimensions when adding new ROI on an image with X/Y arbitrary units (not pixels)
- Improved plot item style serialization management:

  - Before this release, the plot item style was stored in the signal/image metadata only when saving the workspace to an HDF5 file. So, when modifying the style of a signal/image from the "Parameters" button (view toolbar), the style was not kept in some cases (e.g. when duplicating the signal/image).
  - Now, the plot item style is stored in the signal/image metadata whenever the style is modified, and is restored when reloading the workspace

- Handled `ComplexWarning` cast warning when adding regions of interest (ROI) to a signal with complex data

DataLab Version 0.11.0

New features:

- Signals and images may now be reordered in the tree view:
  - Using the new "Move up" and "Move down" actions in the "Edit" menu (or using the corresponding toolbar buttons):
  - This fixes Issue #22 - Add "move up/down" actions in "Edit" menu, for signals/images and groups
- Signals and images may also be reordered using drag and drop:
  - Signals and images can be dragged and dropped inside their own panel to change their order
  - Groups can also be dragged and dropped inside their panel
  - The feature also supports multi-selection (using the standard Ctrl and Shift modifiers), so that multiple signals/images/groups can be moved at once, not necessarily with contiguous positions
  - This fixes Issue #17 - Add Drag and Drop feature to Signals/Images tree views
- New 1D interpolation features:
  - Added "Interpolation" feature to signal panel's "Processing" menu
  - Methods available: linear, spline, quadratic, cubic, barycentric and PCHIP
  - Thanks to @marcel-goldschen-ohm for the contribution to spline interpolation
  - This fixes Issue #20 - Add 1D interpolation features
- New 1D resampling feature:
  - Added "Resampling" feature to signal panel's "Processing" menu
  - Same interpolation methods as for the "Interpolation" feature
  - Possibility to specify the resampling step or the number of points
  - This fixes Issue #21 - Add 1D resampling feature
- New 1D convolution feature:
  - Added "Convolution" feature to signal panel's "Operation" menu
  - This fixes Issue #23 - Add 1D convolution feature
- New 1D detrending feature:
  - Added "Detrending" feature to signal panel's "Processing" menu
  - Methods available: linear or constant
  - This fixes Issue #24 - Add 1D detrending feature
- 2D computing results:

  - Before this release, 2D computing 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 computing results:
  - Additionnaly to the previous enhancement, more computing results are now shown in the "Results" dialog box
  - This concerns both 1D (FHWM, ...) and 2D computing 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 "Computing" menu:
  - This feature allows to plot computing results (1D or 2D)
  - It creates a new signal with X and Y axes corresponding to user-defined parameters (e.g. X = indexes 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 *cdl_testdata.py* plugin accordingly)
- Signal and image computations API (`cdl.core.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 *cdl_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 regions of interest 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 regions of interest 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 `extract_single_roi` function
- Computing 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 - Computing 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 `cdl.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
- Computing result merging:
  - Before this release, when doing a `1->N` computation (sum, average, product) on a group of signals/images, the computing 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 computing 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

DataLab Version 0.10.1

Note: V0.10.0 was almost immediately replaced by V0.10.1 due to a last minute bug fix

New features:

- Features common to signals and images:
  - Added "Real part" and "Imaginary part" features to "Operation" menu
  - Added "Convert data type" feature to "Operation" menu
- Features added following user requests (12/18/2023 meetup @ CEA):
  - Curve and image styles are now saved in the HDF5 file:
    - Curve style covers the following properties: color, line style, line width, marker style, marker size, marker edge color, marker face color, etc.
    - Image style covers the following properties: colormap, interpolation, etc.
    - Those properties were already persistent during the working session, but were lost when saving and reloading the HDF5 file
    - Now, those properties are saved in the HDF5 file and are restored when reloading the HDF5 file
  - New profile extraction features for images:
    - Added "Line profile" to "Operations" menu, to extract a profile from an image along a row or a column
    - Added "Average profile" to "Operations" menu, to extract the average profile on a rectangular area of an image, along a row or a column
  - Image LUT range (contrast/brightness settings) is now saved in the HDF5 file:
    - As for curve and image styles, the LUT range was already persistent during the working session, but was lost when saving and reloading the HDF5 file
    - Now, the LUT range is saved in the HDF5 file and is restored when reloading it
  - Added "Auto-refresh" and "Refresh manually" actions in "View" menu (and main toolbar):
    - When "Auto-refresh" is enabled (default), the plot view is automatically refreshed when a signal/image is modified, added or removed. Even though the refresh is optimized, this may lead to performance issues when working with large datasets.
    - When disabled, the plot view is not automatically refreshed. The user must manually refresh the plot view by clicking on the "Refresh manually" button in the main toolbar or by pressing the standard refresh key (e.g. "F5").
  - Added `toggle_auto_refresh` method to DataLab proxy object:
    - This method allows to toggle the "Auto-refresh" feature from a macro-command, a plugin or a remote control client.
    - A context manager `context_no_refresh` is also available to temporarily disable the "Auto-refresh" feature from a macro-command, a plugin or a remote control client. Typical usage:

```
with proxy.context_no_refresh():
    # Do something without refreshing the plot view
    proxy.compute_fft() # (...)
```

  - Improved curve readability:
    - Until this release, the curve style was automatically set by cycling through **PlotPy** predefined styles

    * However, some styles are not suitable for curve readability (e.g. "cyan" and "yellow" colors are not readable on a white background, especially when combined with a "dashed" line style)
    * This release introduces a new curve style management with colors which are distinguishable and accessible, even to color vision deficiency people

- Added "Curve anti-aliasing" feature to "View" menu (and toolbar):
  - This feature allows to enable/disable curve anti-aliasing (default: enabled)
  - When enabled, the curve rendering is smoother but may lead to performance issues when working with large datasets (that's why it can be disabled)
- Added `toggle_show_titles` method to DataLab proxy object. This method allows to toggle the "Show graphical object titles" feature from a macro-command, a plugin or a remote control client.
- Remote client is now checking the server version and shows a warning message if the server version may not be fully compatible with the client version.

Bug fixes:

- Image contour detection feature ("Computing" 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`

DataLab Version 0.9.2

Bug fixes:

- Region of interest (ROI) extraction feature for images:
  - ROI extraction was not working properly when the "Extract all regions of interest 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

DataLab Version 0.9.1

Bug fixes:

- French translation is not available on Windows/Stand alone version:
  - Locale was not properly detected on Windows for stand-alone version (frozen with `pyinstaller`) due to an issue with `locale.getlocale()` (function returning `None` instead of the expected locale on frozen applications)
  - This is ultimately a `pyinstaller` issue, but a workaround has been implemented in `guidata` V3.2.2 (see guidata issue #68 - Windows: gettext translation is not working on frozen applications)
  - Issue #2 - French translation is not available on Windows Stand alone version
- Saving image to JPEG2000 fails for non integer data:
  - JPEG2000 encoder does not support non integer data or signed integer data
  - Before, DataLab was showing an error message when trying to save incompatible data to JPEG2000: this was not a consistent behavior with other standard image formats (e.g. PNG, JPG, etc.) for which DataLab was automatically converting data to the appropriate format (8-bit unsigned integer)

  - Current behavior is now consistent with other standard image formats: when saving to JPEG2000, DataLab automatically converts data to 8-bit unsigned integer or 16-bit unsigned integer (depending on the original data type)
  - Issue #3 - Save image to JPEG2000: 'OSError: encoder error -2 when writing image file'
- Windows stand-alone version shortcuts not showing in current user start menu:
  - When installing DataLab on Windows from a non-administrator account, the shortcuts were not showing in the current user start menu but in the administrator start menu instead (due to the elevated privileges of the installer and the fact that the installer does not support installing shortcuts for all users)
  - Now, the installer *does not* ask for elevated privileges anymore, and shortcuts are installed in the current user start menu (this also means that the current user must have write access to the installation directory)
  - In future releases, the installer will support installing shortcuts for all users if there is a demand for it (see Issue #5)
  - Issue #4 - Windows: stand-alone version shortcuts not showing in current user start menu
- Installation and configuration window for stand-alone version:
  - Do not show ambiguous error message 'Invalid dependencies' anymore
  - Dependencies are supposed to be checked when building the stand-alone version
- Added PDF documentation to stand-alone version:
  - The PDF documentation was missing in previous release
  - Now, the PDF documentation (in English and French) is included in the stand-alone version

DataLab Version 0.9.0

New dependencies:

- DataLab is now powered by PlotPyStack:
  - PythonQwt
  - guidata
  - PlotPy
- opencv-python (algorithms for image processing)

New reference platform:

- DataLab is validated on Windows 11 with Python 3.11 and PyQt 5.15
- DataLab is also compatible with other OS (Linux, MacOS) and other Python-Qt bindings and versions (Python 3.8-3.12, PyQt6, PySide6)

New features:

- DataLab is a platform:
  - Added support for plugins
    * Custom processing features available in the "Plugins" menu
    * Custom I/O features: new file formats can be added to the standard I/O features for signals and images
    * Custom HDF5 features: new HDF5 file formats can be added to the standard HDF5 import feature
    * More features to come…

  - Added remote control feature: DataLab can be controlled remotely via a TCP/IP connection (see Remote control)
  - Added macro commands: DataLab can be controlled via a macro file (see Macro commands)

- General features:
  - Added settings dialog box (see "Settings" entry in "File" menu):
    * General settings
    * Visualization settings
    * Processing settings
    * Etc.
  - New default layout: signal/image panels are on the right side of the main window, visualization panels are on the left side with a vertical toolbar

- Signal/Image features:
  - Added process isolation: each signal/image is processed in a separate process, so that DataLab does not freeze anymore when processing large signals/images
  - Added support for groups: signals and images can be grouped together, and operations can be applied to all objects in a group, or between groups
  - Added warning and error dialogs with detailed traceback links to the source code (warnings may be optionally ignored)
  - Drastically improved performance when selecting objects
  - Optimized performance when showing large images
  - Added support for dropping files on signal/image panel
  - Added "Computing 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 computing 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 open-source project team.

PYTHON MODULE INDEX

C