
HyperSpy Documentation

Release 2.0.2.dev70+g18caceb

The HyperSpy Developers

Apr 11, 2024

CONTENTS

1	Installing HyperSpy	1
2	Basic Usage	7
3	Loading and saving data	15
4	The Signal class	21
5	Axes handling	55
6	Signal1D Tools	67
7	Signal2D Tools	73
8	Data visualization	79
9	Machine learning	117
10	Model fitting	145
11	Working with big data	173
12	Region Of Interest (ROI)	185
13	Events	193
14	Interactive Operations	197
15	Bibliography	199
16	Gallery of Examples	201
17	Markers	203
18	Signal Creation	205
19	Loading, saving and exporting	207
20	Model fitting	209
21	Region of Interest	211
22	Simple simulations	213

23 Reference	321
24 Getting help	575
25 Changelog	577
26 Contribute	621
27 What is HyperSpy	641
28 HyperSpy's character	643
29 Learning resources	645
30 Citing HyperSpy	647
31 Credits	649
Bibliography	651
Python Module Index	653
Index	655

INSTALLING HYPERSPY

The easiest way to install HyperSpy is to use the *HyperSpy Bundle*, which is available on Windows, MacOS and Linux. Alternatively, HyperSpy can be installed in an existing python distribution, read the *conda installation* and *pip installation* sections for instructions.

Note: To enable the context-menu (right-click) shortcut in a chosen folder, use the `start_jupyter_cm` tool.

Note: If you want to be notified about new releases, please *Watch (Releases only)* the [hyperspy repository on GitHub](#) (requires a [GitHub account](#)).

Warning: Since version 0.8.4 HyperSpy only supports Python 3. If you need to install HyperSpy in Python 2.7 install HyperSpy 0.8.3.

1.1 HyperSpy Bundle

The [HyperSpy](#) bundle is very similar to the Anaconda distribution, and it includes:

- HyperSpy
- HyperSpyUI
- [HyperSpy extensions](#)
- [context menu shortcut \(right-click\)](#) to Jupyter Notebook, Qtconsole or JupyterLab

For instructions, see the [HyperSpy bundle](#) repository.

1.1.1 Portable distribution (Windows only)

A portable version of the [HyperSpy bundle](#) based on the WinPython distribution is also available on Windows.

1.2 Installation using conda

Conda is a package manager for Anaconda-like distributions, such as the [Miniforge](#) or the [HyperSpy-bundle](#). Since HyperSpy is packaged in the [conda-forge](#) channel, it can easily be installed using conda.

To install HyperSpy run the following from the Anaconda Prompt on Windows or from a Terminal on Linux and Mac.

```
$ conda install hyperspy -c conda-forge
```

This will also install the optional GUI packages `hyperspy_gui_ipywidgets` and `hyperspy_gui_traitsui`. To install HyperSpy without the GUI packages, use:

```
$ conda install hyperspy-base -c conda-forge
```

Note: Depending on how Anaconda has been installed, it is possible that the `conda` command is not available from the Terminal, read the [Anaconda User Guide](#) for details.

Note: Using `-c conda-forge` is only necessary when the `conda-forge` channel is not already added to the conda configuration, read the [conda-forge documentation](#) for more details.

Note: Depending on the packages installed in Anaconda, `conda` can be slow and in this case `mamba` can be used as an alternative of `conda` since the former is significantly faster. Read the [mamba documentation](#) for instructions.

1.2.1 Further information

When installing packages, `conda` will verify that all requirements of *all* packages installed in an environment are met. This can lead to situations where a solution for dependencies resolution cannot be resolved or the solution may include installing old or undesired versions of libraries. The requirements depend on which libraries are already present in the environment as satisfying their respective dependencies may be problematic. In such a situation, possible solutions are:

- use Miniconda instead of Anaconda, if you are installing a python distribution from scratch: Miniconda only installs very few packages so satisfying all dependencies is simple.
- install HyperSpy in a [new environment](#). The following example illustrates how to create a new environment named `hspy_environment`, activate it and install HyperSpy in the new environment.

```
$ conda create -n hspy_environment
$ conda activate hspy_environment
$ conda install hyperspy -c conda-forge
```

Note: A consequence of installing `hyperspy` in a new environment is that you need to activate this environment using `conda activate environment_name` where `environment_name` is the name of the environment, however *shortcuts* can be created using different approaches:

- Install `start_jupyter_cm` in the `hyperspy` environment.
- Install `nb_conda_kernels`.
- Create IPython kernels for different environment.

To learn more about the Anaconda eco-system:

- Choose between [Anaconda](#) or [Miniconda](#)?
- Understanding [conda](#) and [pip](#).
- What is [conda-forge](#).

1.3 Installation using pip

HyperSpy is listed in the [Python Package Index](#). Therefore, it can be automatically downloaded and installed [pip](#). You may need to install [pip](#) for the following commands to run.

To install all of HyperSpy's functionalities, run:

```
$ pip install hyperspy[all]
```

To install only the strictly required dependencies and limited functionalities, use:

```
$ pip install hyperspy
```

See the following list of selectors to select the installation of optional dependencies required by specific functionalities:

- `ipython` for integration with the *ipython* terminal and parallel processing using *ipyparallel*,
- `learning` for some machine learning features,
- `gui-jupyter` to use the [Jupyter widgets](#) GUI elements,
- `gui-traitsui` to use the GUI elements based on [traitsui](#),
- `speed` install `numba` and `numexpr` to speed up some functionalities,
- `tests` to install required libraries to run HyperSpy's unit tests,
- `coverage` to coverage statistics when running the tests,
- `doc` to install required libraries to build HyperSpy's documentation,
- `dev` to install all the above,
- `all` to install all the above except the development requirements (`tests`, `doc` and `dev`).

For example:

```
$ pip install hyperspy[learning, gui-jupyter]
```

Finally, be aware that HyperSpy depends on a number of libraries that usually need to be compiled and therefore installing HyperSpy may require development tools installed in the system. If the above does not work for you remember that the easiest way to install HyperSpy is [using the HyperSpy bundle](#).

1.4 Update HyperSpy

1.4.1 Using conda

To update hyperspy to the latest release using conda:

```
$ conda update hyperspy -c conda-forge
```

1.4.2 Using pip

To update hyperspy to the latest release using pip:

```
$ pip install hyperspy --upgrade
```

1.5 Install specific version

1.5.1 Using conda

To install a specific version of hyperspy (for example 1.6.1) using conda:

```
$ conda install hyperspy=1.6.1 -c conda-forge
```

1.5.2 Using pip

To install a specific version of hyperspy (for example 1.6.1) using pip:

```
$ pip install hyperspy==1.6.1
```

1.6 Rolling release Linux distributions

Due to the requirement of up to date versions for dependencies such as *numpy*, *scipy*, etc., binary packages of HyperSpy are not provided for most linux distributions and the installation via *Anaconda/Miniconda* or *Pip* is recommended.

However, packages of the latest HyperSpy release and the related GUI packages are maintained for the rolling release distributions *Arch-Linux* (in the *Arch User Repository*) (AUR) and *openSUSE* (*Community Package*) as *python-hyperspy* and *python-hyperspy-gui-traitsui*, *python-hyperspy-gui-ipywidgets* for the GUIs packages.

A more up-to-date package that contains all updates to be included in the next minor version release (likely including new features compared to the stable release) is also available in the AUR as *python-hyperspy-git*.

1.7 Install development version

1.7.1 Clone the hyperspy repository

To get the development version from our git repository you need to install `git`. Then just do:

```
$ git clone https://github.com/hyperspy/hyperspy.git
```

Warning: When running hyperspy from a development version, it can happen that the dependency requirement changes in which you will need to keep this this requirement up to date (check dependency requirement in `setup.py`) or run again the installation in development mode using `pip` as explained below.

1.7.2 Installation in a Anaconda/Miniconda distribution

Optionally, create an environment to separate your hyperspy installation from other anaconda environments ([read more about environments here](#)):

```
$ conda create -n hspy_dev python # create an empty environment with latest python
$ conda activate hspy_dev # activate environment
```

Install the runtime and development dependencies requirements using conda:

```
$ conda install hyperspy-base -c conda-forge --only-deps # install hyperspy dependencies
$ conda install hyperspy-dev -c conda-forge # install developer dependencies
```

The package `hyperspy-dev` will install the development dependencies required for testing and building the documentation.

From the root folder of your hyperspy repository (folder containing the `setup.py` file) run `pip` in development mode:

```
$ pip install -e . --no-deps # install the currently checked-out branch of hyperspy
```

1.7.3 Installation in other (non-system) Python distribution

From the root folder of your hyperspy repository (folder containing the `setup.py` file) run `pip` in development mode:

```
$ pip install -e .[dev]
```

All required dependencies are automatically installed by `pip`. If you don't want to install all dependencies and only install some of the optional dependencies, use the corresponding selector as explained in the [Installation using pip](#) section

1.7.4 Installation in a system Python distribution

When using a system Python distribution, it is recommended to install the dependencies using your system package manager.

From the root folder of your hyperspy repository (folder containing the `setup.py` file) run `pip` in development mode.

```
$ pip install -e --user .[dev]
```

1.7.5 Creating Debian/Ubuntu binaries

You can create binaries for Debian/Ubuntu from the source by running the `release_debian` script

```
$ ./release_debian
```

Warning: For this to work, the following packages must be installed in your system `python-stdeb`, `dehelper`, `dpkg-dev` and `python-argparser` are required.

BASIC USAGE

2.1 Starting Python in Windows

If you used the bundle installation you should be able to use the context menus to get started. Right-click on the folder containing the data you wish to analyse and select “Jupyter notebook here” or “Jupyter qtconsole here”. We recommend the former, since notebooks have many advantages over conventional consoles, as will be illustrated in later sections. The examples in some later sections assume Notebook operation. A new tab should appear in your default browser listing the files in the selected folder. To start a python notebook choose “Python 3” in the “New” drop-down menu at the top right of the page. Another new tab will open which is your Notebook.

2.2 Starting Python in Linux and MacOS

You can start IPython by opening a system terminal and executing `ipython`, (optionally followed by the “frontend”: “qtconsole” for example). However, in most cases, **the most agreeable way** to work with HyperSpy interactively is using the [Jupyter Notebook](#) (previously known as the IPython Notebook), which can be started as follows:

```
$ jupyter notebook
```

Linux users may find it more convenient to start Jupyter/IPython from the [file manager context menu](#). In either OS you can also start by [double-clicking a notebook file](#) if one already exists.

2.3 Starting HyperSpy in the notebook (or terminal)

Typically you will need to [set up IPython for interactive plotting with matplotlib](#) using `%matplotlib` (which is known as a ‘Jupyter magic’) *before executing any plotting command*. So, typically, after starting IPython, you can import HyperSpy and set up interactive matplotlib plotting by executing the following two lines in the IPython terminal (In these docs we normally use the general Python prompt symbol `>>>` but you will probably see `In [1]:` etc.):

```
>>> %matplotlib qt
>>> import hyperspy.api as hs
```

Note that to execute lines of code in the notebook you must press `Shift+Return`. (For details about notebooks and their functionality try the help menu in the notebook). Next, import two useful modules: `numpy` and `matplotlib.pyplot`, as follows:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

The rest of the documentation will assume you have done this. It also assumes that you have installed at least one of HyperSpy's GUI packages: `jupyter widgets` GUI and the `traitsui` GUI.

2.3.1 Possible warnings when importing HyperSpy?

HyperSpy supports different GUIs and `matplotlib` backends which in specific cases can lead to warnings when importing HyperSpy. Most of the time there is nothing to worry about — the warnings simply inform you of several choices you have. There may be several causes for a warning, for example:

- not all the GUIs packages are installed. If none is installed, we recommend you to install at least the `hyperspy-gui-ipywidgets` package if you are planning to perform interactive data analysis in the Jupyter Notebook. Otherwise, you can simply disable the warning in *preferences* as explained below.
- the `hyperspy-gui-traitsui` package is installed and you are using an incompatible `matplotlib` backend (e.g. `notebook`, `nbagg` or `widget`).
 - If you want to use the `traitsui` GUI, use the `qt` `matplotlib` backend instead.
 - Alternatively, if you prefer to use the `notebook` or `widget` `matplotlib` backend, and if you don't want to see the (harmless) warning, make sure that you have the `hyperspy-gui-ipywidgets` installed and disable the `traitsui` GUI in the *preferences*.

Changed in version v1.3: HyperSpy works with all `matplotlib` backends, including the `notebook` (also called `nbAgg`) backend that enables interactive plotting embedded in the jupyter notebook.

Note: When running in a headless system it is necessary to set the `matplotlib` backend appropriately to avoid a *cannot connect to X server* error, for example as follows:

```
>>> import matplotlib
>>> matplotlib.rcParams["backend"] = "Agg"
>>> import hyperspy.api as hs
```

2.4 Getting help

When using IPython, the documentation (docstring in Python jargon) can be accessed by adding a question mark to the name of a function. e.g.:

```
In [1]: import hyperspy.api as hs
```

This syntax is a shortcut to the standard way one of displaying the help associated to a given functions (docstring in Python jargon) and it is one of the many features of IPython, which is the interactive python shell that HyperSpy uses under the hood.

2.5 Autocompletion

Another useful IPython feature is the [autocompletion](#) of commands and filenames using the tab and arrow keys. It is highly recommended to read the [IPython introduction](#) for many more useful features that will boost your efficiency when working with HyperSpy/Python interactively.

2.6 Creating signal from a numpy array

HyperSpy can operate on any numpy array by assigning it to a BaseSignal class. This is useful e.g. for loading data stored in a format that is not yet supported by HyperSpy—supposing that they can be read with another Python library—or to explore numpy arrays generated by other Python libraries. Simply select the most appropriate signal from the [signals](#) module and create a new instance by passing a numpy array to the constructor e.g.

```
>>> my_np_array = np.random.random((10, 20, 100))
>>> s = hs.signals.Signal1D(my_np_array)
>>> s
<Signal1D, title: , dimensions: (20, 10|100)>
```

The numpy array is stored in the `data` attribute of the signal class:

```
>>> s.data
```

2.7 The navigation and signal dimensions

In HyperSpy the data is interpreted as a signal array and, therefore, the data axes are not equivalent. HyperSpy distinguishes between *signal* and *navigation* axes and most functions operate on the *signal* axes and iterate on the *navigation* axes. For example, an EELS spectrum image (i.e. a 2D array of spectra) has three dimensions X, Y and energy-loss. In HyperSpy, X and Y are the *navigation* dimensions and the energy-loss is the *signal* dimension. To make this distinction more explicit the representation of the object includes a separator | between the navigation and signal dimensions e.g.

In HyperSpy a spectrum image has signal dimension 1 and navigation dimension 2 and is stored in the Signal1D subclass.

```
>>> s = hs.signals.Signal1D(np.zeros((10, 20, 30)))
>>> s
<Signal1D, title: , dimensions: (20, 10|30)>
```

An image stack has signal dimension 2 and navigation dimension 1 and is stored in the Signal2D subclass.

```
>>> im = hs.signals.Signal2D(np.zeros((30, 10, 20)))
>>> im
<Signal2D, title: , dimensions: (30|20, 10)>
```

Note that HyperSpy rearranges the axes when compared to the array order. The following few paragraphs explain how and why it does it.

Depending how the array is arranged, some axes are faster to iterate than others. Consider an example of a book as the dataset in question. It is trivially simple to look at letters in a line, and then lines down the page, and finally pages in the whole book. However if your words are written vertically, it can be inconvenient to read top-down (the lines are still horizontal, it's just the meaning that's vertical!). It's very time-consuming if every letter is on a different page, and for

every word you have to turn 5-6 pages. Exactly the same idea applies here - in order to iterate through the data (most often for plotting, but applies for any other operation too), you want to keep it ordered for “fast access”.

In Python (more explicitly *numpy*) the “fast axes order” is C order (also called row-major order). This means that the **last** axis of a numpy array is fastest to iterate over (i.e. the lines in the book). An alternative ordering convention is F order (column-major), where it is the reverse - the first axis of an array is the fastest to iterate over. In both cases, the further an axis is from the *fast axis* the slower it is to iterate over it. In the book analogy you could think, for example, think about reading the first lines of all pages, then the second and so on.

When data is acquired sequentially it is usually stored in acquisition order. When a dataset is loaded, HyperSpy generally stores it in memory in the same order, which is good for the computer. However, HyperSpy will reorder and classify the axes to make it easier for humans. Let’s imagine a single numpy array that contains pictures of a scene acquired with different exposure times on different days. In numpy the array dimensions are (D, E, Y, X). This order makes it fast to iterate over the images in the order in which they were acquired. From a human point of view, this dataset is just a collection of images, so HyperSpy first classifies the image axes (X and Y) as *signal axes* and the remaining axes the *navigation axes*. Then it reverses the order of each sets of axes because many humans are used to get the X axis first and, more generally the axes in acquisition order from left to right. So, the same axes in HyperSpy are displayed like this: (E, D | X, Y).

Extending this to arbitrary dimensions, by default, we reverse the numpy axes, chop it into two chunks (signal and navigation), and then swap those chunks, at least when printing. As an example:

```
(a1, a2, a3, a4, a5, a6) # original (numpy)
(a6, a5, a4, a3, a2, a1) # reverse
(a6, a5) (a4, a3, a2, a1) # chop
(a4, a3, a2, a1) (a6, a5) # swap (HyperSpy)
```

In the background, HyperSpy also takes care of storing the data in memory in a “machine-friendly” way, so that iterating over the navigation axes is always fast.

2.8 Saving Files

The data can be saved to several file formats. The format is specified by the extension of the filename.

```
>>> # load the data
>>> d = hs.load("example.tif")
>>> # save the data as a tiff
>>> d.save("example_processed.tif")
>>> # save the data as a png
>>> d.save("example_processed.png")
>>> # save the data as an hspy file
>>> d.save("example_processed.hspy")
```

Some file formats are much better at maintaining the information about how you processed your data. The preferred formats are *hspy* and *zspy*, because they are open formats and keep most information possible.

There are optional flags that may be passed to the save function. See *Saving* for more details.

2.9 Accessing and setting the metadata

When loading a file HyperSpy stores all metadata in the BaseSignal *original_metadata* attribute. In addition, some of those metadata and any new metadata generated by HyperSpy are stored in *metadata* attribute.

```
>>> import exspy
>>> s = exspy.data.eelsdb(formula="NbO2", edge="M2,3")[0]
>>> s.metadata
├── Acquisition_instrument
│   └── TEM
│       ├── Detector
│       │   └── EELS
│       │       └── collection_angle = 6.5
│       ├── beam_energy = 100.0
│       ├── convergence_angle = 10.0
│       └── microscope = VG HB501UX
├── General
│   ├── author = Wilfried Sigle
│   └── title = Niobium oxide NbO2
├── Sample
│   ├── chemical_formula = NbO2
│   ├── description = Analyst: David Bach, Wilfried Sigle. Temperature: Room.
│   └── elements = ['Nb', 'O']
└── Signal
    ├── quantity = Electrons ()
    └── signal_type = EELS

>>> s.original_metadata
├── emsa
│   ├── DATATYPE = XY
│   ├── DATE =
│   ├── FORMAT = EMSA/MAS Spectral Data File
│   ├── NCOLUMNS = 1.0
│   ├── NPOINTS = 1340.0
│   ├── OFFSET = 120.0003
│   ├── OWNER = eelsdatabase.net
│   ├── SIGNALTYPE = ELS
│   ├── TIME =
│   ├── TITLE = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217
│   ├── VERSION = 1.0
│   ├── XPERCHAN = 0.5
│   ├── XUNITS = eV
│   └── YUNITS =
└── json
    ├── api_permalink = https://api.eelsdb.eu/spectra/niobium-oxide-nbo2-2/
    ├── associated_spectra = [{'name': 'Niobium oxide NbO2', 'link': 'https://eelsdb.eu/
    ↪spectra/niobium-oxide-nbo2/', 'type': 'Low Loss'}]
    ├── author
    │   ├── name = Wilfried Sigle
    │   ├── profile_api_url = https://api.eelsdb.eu/author/wsigle/
    │   └── profile_url = https://eelsdb.eu/author/wsigle/
    ├── beamenergy = 100 kV
    └── collection = 6.5 mrad
```

(continues on next page)

(continued from previous page)

```

— comment_count = 0
— convergence = 10 mrad
— darkcurrent = Yes
— description = Analyst: David Bach, Wilfried Sigle. Temperature: Room.
— detector = Parallel: Gatan ENFINA
— download_link = https://eelsdb.eu/wp-content/uploads/2015/09/DspecYB7EbW.msa
— edges = ['Nb_M2,3', 'Nb_M4,5', 'O_K']
— elements = ['Nb', 'O']
— formula = NbO2
— gainvariation = Yes
— guntype = cold field emission
— id = 21727
— integratetime = 5 secs
— keywords = ['imported from old site']
— max_energy = 789.5 eV
— microscope = VG HB501UX
— min_energy = 120 eV
— monochromated = No
— other_links = [{'url': 'http://pc-web.cemes.fr/eelsdb/index.php?page=displayspec.
↳php&id=217', 'title': 'Old EELS DB'}]
— permalink = https://eelsdb.eu/spectra/niobium-oxide-nbo2-2/
— published = 2008-02-15 00:00:00
— readouts = 10
— resolution = 1.3 eV
— stepSize = 0.5 eV/pixel
— thickness = 0.58 t/λ;
— title = Niobium oxide NbO2
— type = Core Loss

>>> s.metadata.General.title = "NbO2 Nb_M edge"
>>> s.metadata
— Acquisition_instrument
  — TEM
    — Detector
      — EELS
        — collection_angle = 6.5
    — beam_energy = 100.0
    — convergence_angle = 10.0
    — microscope = VG HB501UX
— General
  — author = Wilfried Sigle
  — title = NbO2 Nb_M edge
— Sample
  — chemical_formula = NbO2
  — description = Analyst: David Bach, Wilfried Sigle. Temperature: Room.
  — elements = ['Nb', 'O']
— Signal
  — quantity = Electrons ( )
  — signal_type = EELS

```

2.10 Configuring HyperSpy

The behaviour of HyperSpy can be customised using the preferences. The easiest way to do it is by calling the `gui()` method:

```
>>> hs.preferences.gui()
```

This command should raise the Preferences user interface if one of the hyperspy gui packages are installed and enabled:

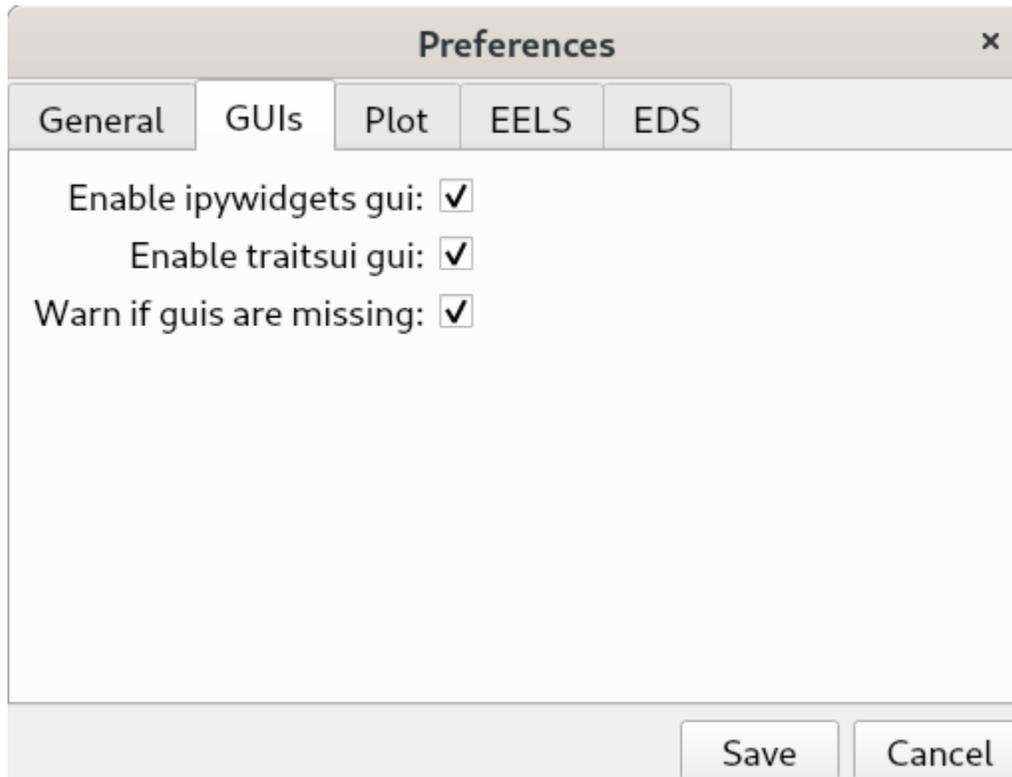


Fig. 1: Preferences user interface.

New in version 1.3: Possibility to enable/disable GUIs in the preferences.

It is also possible to set the preferences programmatically. For example, to disable the traitsui GUI elements and save the changes to disk:

```
>>> hs.preferences.GUIs.enable_traitsui_gui = False
>>> hs.preferences.save()
>>> # if not saved, this setting will be used until the next jupyter kernel shutdown
```

Changed in version 1.3: The following items were removed from preferences: `General.default_export_format`, `General.lazy`, `Model.default_fitter`, `Machine_learning.multiple_files`, `Machine_learning.same_window`, `Plot.default_style_to_compare_spectra`, `Plot.plot_on_load`, `Plot.pylab_inline`, `EELS.fine_structure_width`, `EELS.fine_structure_active`, `EELS.fine_structure_smoothing`, `EELS.synchronize_cl_with_ll`, `EELS.preedge_safe_window_width`, `EELS.min_distance_between_edges_for_fine_structure`.

2.11 Messages log

HyperSpy writes messages to the [Python logger](#). The default log level is “WARNING”, meaning that only warnings and more severe event messages will be displayed. The default can be set in the *preferences*. Alternatively, it can be set using `set_log_level()` e.g.:

```
>>> import hyperspy.api as hs
>>> hs.set_log_level('INFO')
>>> hs.load('my_file.dm3')
INFO:hyperspy.io_plugins.digital_micrograph:DM version: 3
INFO:hyperspy.io_plugins.digital_micrograph:size 4796607 B
INFO:hyperspy.io_plugins.digital_micrograph:Is file Little endian? True
INFO:hyperspy.io_plugins.digital_micrograph:Total tags in root group: 15
<Signal2D, title: My file, dimensions: (|1024, 1024)
```

LOADING AND SAVING DATA

Changed in version 2.0: The IO plugins formerly developed within HyperSpy have been moved to the separate package [RosettaSciIO](#) in order to facilitate a wider use also by other packages. Plugins supporting additional formats or corrections/enhancements to existing plugins should now be contributed to the [RosettaSciIO repository](#) and file format specific issues should be reported to the [RosettaSciIO issue tracker](#).

3.1 Loading

3.1.1 Basic usage

HyperSpy can read and write to multiple formats (see [Supported formats](#)). To load data use the `load()` command. For example, to load the image `spam.jpg`, you can type:

```
>>> s = hs.load("spam.jpg")
```

If loading was successful, the variable `s` contains a HyperSpy signal or any type of signal defined in one of the *HyperSpy extensions*, see [Specifying signal type](#) for more details.

Note: When the file contains several datasets, the `load()` function will return a list of HyperSpy signals, instead of a single HyperSpy signal. Each signal can then be accessed using list indexing.

```
>>> s = hs.load("spameggsandham.hspy")
>>> s
[<Signal1D, title: spam, dimensions: (32,32|1024)>,
 <Signal1D, title: eggs, dimensions: (32,32|1024)>,
 <Signal1D, title: ham, dimensions: (32,32|1024)>]
```

Using indexing to access the first signal (index 0):

```
>>> s[0]
<Signal1D, title: spam, dimensions: (32,32|1024)>
```

Hint: The load function returns an object that contains data read from the file. We assign this object to the variable `s` but you can choose any (valid) variable name you like. for the filename, don't forget to include the quotation marks and the file extension.

If no argument is passed to the load function, a window will be raised that allows to select a single file through your OS file manager, e.g.:

```
>>> # This raises the load user interface
>>> s = hs.load()
```

It is also possible to load multiple files at once or even stack multiple files. For more details read [Loading multiple files](#).

3.1.2 Specifying reader

HyperSpy will attempt to infer the appropriate file reader to use based on the file extension (for example. `.hspy`, `.emd` and so on). You can override this using the `reader` keyword:

```
# Load a .hspy file with an unknown extension
>>> s = hs.load("filename.some_extension", reader="hspy") # doctest: +SKIP
```

3.1.3 Specifying signal type

HyperSpy will attempt to infer the most suitable signal type for the data being loaded. Domain specific signal types are provided by [extension libraries](#). To list the signal types available on your local installation use:

```
>>> hs.print_known_signal_types()
```

When loading data, the signal type can be specified by providing the `signal_type` keyword, which has to correspond to one of the available subclasses of signal:

```
>>> s = hs.load("filename", signal_type="EELS")
```

If the loaded file contains several datasets, the `load()` function will return a list of the corresponding signals:

```
>>> s = hs.load("spameggsandham.hspy")
>>> s
[<Signal1D, title: spam, dimensions: (32,32|1024)>,
 <Signal1D, title: eggs, dimensions: (32,32|1024)>,
 <Signal1D, title: ham, dimensions: (32,32|1024)>]
```

Note: Note for python programmers: the data is stored in a numpy array in the `data` attribute, but you will not normally need to access it there.

3.1.4 Metadata

Most scientific file formats store some extra information about the data and the conditions under which it was acquired (metadata). HyperSpy reads most of them and stores them in the `original_metadata` attribute. Also, depending on the file format, a part of this information will be mapped by HyperSpy to the `metadata` attribute, where it can for example be used by routines operating on the signal. See the [metadata structure](#) for details.

Note: Extensive metadata can slow down loading and processing, and loading the `original_metadata` can be disabled using the `load_original_metadata` argument of the `load()` function. If this argument is set to `False`, the `metadata` will still be populated.

To print the content of the attributes simply use:

```
>>> s.original_metadata
>>> s.metadata
```

The `original_metadata` and `metadata` can be exported to text files using the `export()` method, e.g.:

```
>>> s.original_metadata.export('parameters')
```

3.1.5 Lazy loading of large datasets

New in version 1.2: lazy keyword argument.

Almost all file readers support *lazy* loading, which means accessing the data without loading it to memory (see [Supported formats](#) for a list). This feature can be useful when analysing large files. To use this feature, set `lazy` to `True` e.g.:

```
>>> s = hs.load("filename.hspy", lazy=True)
```

More details on lazy evaluation support can be found in *Working with big data*.

The units of the navigation and signal axes can be converted automatically during loading using the `convert_units` parameter. If `True`, the `convert_to_units` method of the `axes_manager` will be used for the conversion and if set to `False`, the units will not be converted (default).

3.1.6 Loading multiple files

Rather than loading files individually, several files can be loaded with a single command. This can be done by passing a list of filenames to the load functions, e.g.:

```
>>> s = hs.load(["file1.hspy", "file2.hspy"])
```

or by using shell-style wildcards:

```
>>> s = hs.load("file*.hspy")
```

Alternatively, regular expression type character classes can be used such as `[a-z]` for lowercase letters or `[0-9]` for one digit integers:

```
>>> s = hs.load('file[0-9].hspy')
```

Note: Wildcards are implemented using `glob.glob()`, which treats `*`, `[` and `]` as special characters for pattern matching. If your filename or path contains square brackets, you may want to set `escape_square_brackets=True`:

```
>>> # Say there are two files like this:
>>> # /home/data/afile[1x1].hspy
>>> # /home/data/afile[1x2].hspy

>>> s = hs.load("/home/data/afile[*].hspy", escape_square_brackets=True)
```

HyperSpy also supports `pathlib.Path` <https://docs.python.org/3/library/pathlib.html> objects, for example:

```
>>> import hyperspy.api as hs
>>> from pathlib import Path

>>> # Use pathlib.Path
>>> p = Path("/path/to/a/file.hspy")
>>> s = hs.load(p)

>>> # Use pathlib.Path.glob
>>> p = Path("/path/to/some/files/").glob("*.hspy")
>>> s = hs.load(p)
```

By default HyperSpy will return a list of all the files loaded. Alternatively, by setting `stack=True`, HyperSpy can be instructed to stack the data - given that the files contain data with exactly the same dimensions. If this is not the case, an error is raised. If each file contains multiple (N) signals, N stacks will be created. Here, the number of signals per file must also match, or an error will be raised.

```
>>> ls
CL1.raw CL1.rpl CL2.raw CL2.rpl CL3.raw CL3.rpl CL4.raw CL4.rpl
LL3.raw LL3.rpl shift_map-SI3.npy hdf5/
>>> s = hs.load('*.rpl')
>>> s
[<EELSSpectrum, title: CL1, dimensions: (64, 64, 1024)>,
<EELSSpectrum, title: CL2, dimensions: (64, 64, 1024)>,
<EELSSpectrum, title: CL3, dimensions: (64, 64, 1024)>,
<EELSSpectrum, title: CL4, dimensions: (64, 64, 1024)>,
<EELSSpectrum, title: LL3, dimensions: (64, 64, 1024)>]
>>> s = hs.load('*.rpl', stack=True)
>>> s
<EELSSpectrum, title: mva, dimensions: (5, 64, 64, 1024)>
```

3.1.7 Loading example data and data from online databases

HyperSpy is distributed with some example data that can be found in *data*:

```
>>> s = hs.data.two_gaussians()
>>> s.plot()
```

New in version 1.4: *data* (formerly `hyperspy.api.datasets.artificial_data`)

There are also artificial datasets, which are made to resemble real experimental data.

```
>>> s = hs.data.atomic_resolution_image()
>>> s.plot()
```

3.2 Saving

To save data to a file use the `save()` method. The first argument is the filename and the format is defined by the filename extension. If the filename does not contain the extension, the default format (**HSpy-HDF5**) is used. For example, if the `s` variable contains the *BaseSignal* that you want to write to a file, the following will write the data to a file called `spectrum.hspy` in the default **HSpy-HDF5** format:

```
>>> s.save('spectrum')
```

If you want to save to the **ripple** format instead, write:

```
>>> s.save('spectrum.rpl')
```

Some formats take extra arguments. See the corresponding pages at [Supported formats](#) for more information.

THE SIGNAL CLASS

Warning: This subsection can be a bit confusing for beginners. Do not worry if you do not understand it all.

HyperSpy stores the data in the *BaseSignal* class, that is the object that you get when e.g. you load a single file using *load()*. Most of the data analysis functions are also contained in this class or its specialized subclasses. The *BaseSignal* class contains general functionality that is available to all the subclasses. The subclasses provide functionality that is normally specific to a particular type of data, e.g. the *Signal1D* class provides common functionality to deal with one-dimensional (e.g. spectral) data and *exspy.signals.EELSSpectrum* (which is a subclass of *Signal1D*) adds extra functionality to the *Signal1D* class for electron energy-loss spectroscopy data analysis.

A signal store other objects in what are called attributes. For examples, the data is stored in a numpy array in the *data* attribute, the original parameters in the *original_metadata* attribute, the mapped parameters in the *metadata* attribute and the axes information (including calibration) can be accessed (and modified) in the *AxesManager* attribute.

4.1 Basics of signals

4.1.1 Signal initialization

Many of the values in the *AxesManager* can be set when making the *BaseSignal* object.

```
>>> dict0 = {'size': 10, 'name':'Axis0', 'units':'A', 'scale':0.2, 'offset':1}
>>> dict1 = {'size': 20, 'name':'Axis1', 'units':'B', 'scale':0.1, 'offset':2}
>>> s = hs.signals.BaseSignal(np.random.random((10,20)), axes=[dict0, dict1])
>>> s.axes_manager
<Axes manager, axes: (|20, 10)>
```

Name	size	index	offset	scale	units
Axis1	20	0	2	0.1	B
Axis0	10	0	1	0.2	A

This also applies to the *metadata*.

```
>>> metadata_dict = {'General':{'name':'A BaseSignal'}}
>>> metadata_dict['General']['title'] = 'A BaseSignal title'
>>> s = hs.signals.BaseSignal(np.arange(10), metadata=metadata_dict)
>>> s.metadata
├─ General
```

(continues on next page)

(continued from previous page)

```

├── name = A BaseSignal
├── title = A BaseSignal title
└── Signal
    ├── signal_type =

```

Instead of using a list of *axes dictionaries* [`dict0`, `dict1`] during signal initialization, you can also pass a list of *axes objects*: [`axis0`, `axis1`].

4.1.2 The navigation and signal dimensions

HyperSpy can deal with data of arbitrary dimensions. Each dimension is internally classified as either “navigation” or “signal” and the way this classification is done determines the behaviour of the signal.

The concept is probably best understood with an example: let’s imagine a three dimensional dataset e.g. a numpy array with dimensions (10, 20, 30). This dataset could be an spectrum image acquired by scanning over a sample in two dimensions. As in this case the signal is one-dimensional we use a *Signal1D* subclass for this data e.g.:

```

>>> s = hs.signals.Signal1D(np.random.random((10, 20, 30)))
>>> s
<Signal1D, title: , dimensions: (20, 10|30)>

```

In HyperSpy’s terminology, the *signal dimension* of this dataset is 30 and the navigation dimensions (20, 10). Notice the separator | between the navigation and signal dimensions.

However, the same dataset could also be interpreted as an image stack instead. Actually it could has been acquired by capturing two dimensional images at different wavelengths. Then it would be natural to identify the two spatial dimensions as the signal dimensions and the wavelength dimension as the navigation dimension. To view the data in this way we could have used a *Signal2D* instead e.g.:

```

>>> im = hs.signals.Signal2D(np.random.random((10, 20, 30)))
>>> im
<Signal2D, title: , dimensions: (10|30, 20)>

```

Indeed, for data analysis purposes, one may like to operate with an image stack as if it was a set of spectra or viceversa. One can easily switch between these two alternative ways of classifying the dimensions of a three-dimensional dataset by *transforming between BaseSignal subclasses*.

The same dataset could be seen as a three-dimensional signal:

```

>>> td = hs.signals.BaseSignal(np.random.random((10, 20, 30)))
>>> td
<BaseSignal, title: , dimensions: (|30, 20, 10)>

```

Notice that with use *BaseSignal* because there is no specialised subclass for three-dimensional data. Also note that by default *BaseSignal* interprets all dimensions as signal dimensions. We could also configure it to operate on the dataset as a three-dimensional array of scalars by changing the default *view* of *BaseSignal* by taking the transpose of it:

```

>>> scalar = td.T
>>> scalar
<BaseSignal, title: , dimensions: (30, 20, 10|)>

```

For more examples of manipulating signal axes in the “signal-navigation” space can be found in *Transposing (changing signal spaces)*.

Note: Although each dimension can be arbitrarily classified as “navigation dimension” or “signal dimension”, for most common tasks there is no need to modify HyperSpy’s default choice.

4.1.3 Signal subclasses

The `signals` module, which contains all available signal subclasses, is imported in the user namespace when loading HyperSpy. In the following example we create a `Signal2D` instance from a 2D numpy array:

```
>>> im = hs.signals.Signal2D(np.random.random((64,64)))
>>> im
<Signal2D, title: , dimensions: (|64, 64)>
```

The table below summarises all the `BaseSignal` subclasses currently distributed with HyperSpy. From HyperSpy 2.0, all domain specific signal subclasses, characterized by the `signal_type` metadata attribute, are provided by dedicated *extension packages*.

The generic subclasses provided by HyperSpy are characterized by the the data `dtype` and the signal dimension. In particular, there are specialised signal subclasses to handle complex data. See the table and diagram below. Where appropriate, functionalities are restricted to certain `BaseSignal` subclasses.

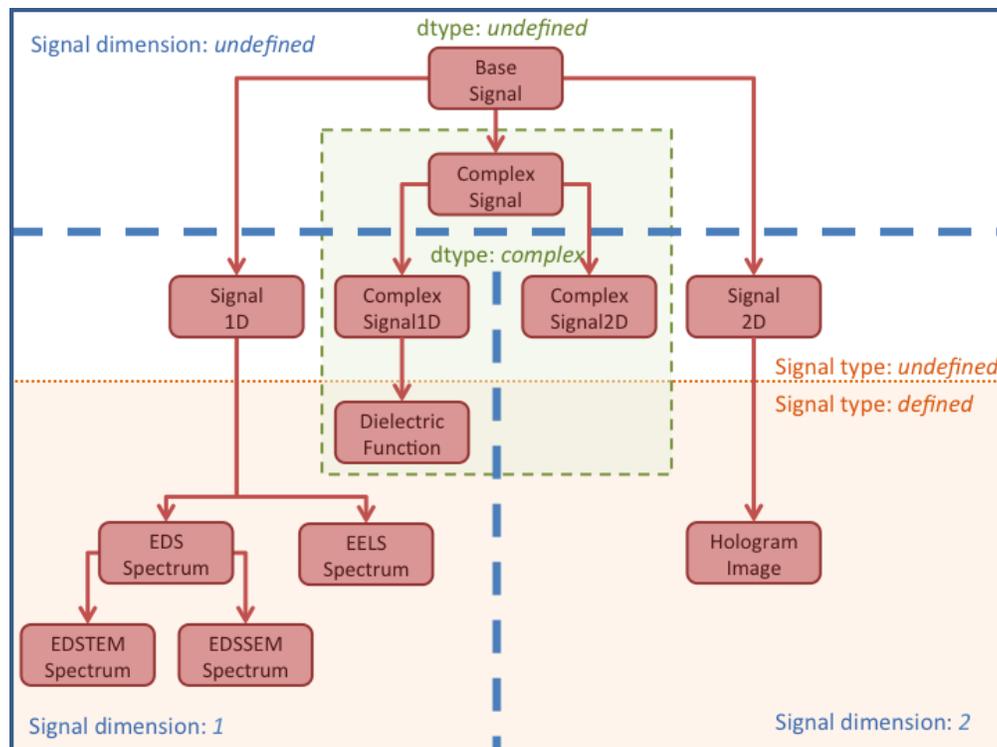


Fig. 1: Diagram showing the inheritance structure of the different subclasses. The upper part contains the generic classes shipped with HyperSpy. The lower part contains examples of domain specific subclasses provided by some of the *HyperSpy extensions*.

Table 1: BaseSignal subclass characteristics.

BaseSignal subclass	signal_dimension	signal_type	dtype
<i>BaseSignal</i>	.	.	real
<i>Signal1D</i>	1	.	real
<i>Signal2D</i>	2	.	real
<i>ComplexSignal</i>	.	.	complex
<i>ComplexSignal1D</i>	1	.	complex
<i>ComplexSignal2D</i>	2	.	complex

Changed in version 1.0: The subclasses `Simulation`, `SpectrumSimulation` and `ImageSimulation` were removed.

New in version 1.5: External packages can register extra *BaseSignal* subclasses.

Changed in version 2.0: The subclasses `EELS`, `EDS_SEM`, `EDS_TEM` and `DielectricFunction` have been moved to the extension package `ElEXSpy` and the subclass `hologram` has been moved to the extension package `HoloSpy`.

4.1.4 HyperSpy extensions

Domain specific functionalities for specific types of data are provided through a number of dedicated python packages that qualify as *HyperSpy extensions*. These packages provide subclasses of the generic signal classes listed above, depending on the dimensionality and type of the data. Some examples are included in the *diagram above*. If an extension package is installed on your system, the provided signal subclasses are registered with HyperSpy and these classes are directly available when loading the `hyperspy.api` into the namespace. A [list of packages that extend HyperSpy](#) is curated in a dedicated repository.

The metadata attribute `signal_type` describes the nature of the signal. It can be any string, normally the acronym associated with a particular signal. To print all *BaseSignal* subclasses available in your system call the function `print_known_signal_types()` as in the following example:

```
>>> hs.print_known_signal_types()
+-----+-----+-----+-----+
| signal_type | aliases | class name | package |
+-----+-----+-----+-----+
| DielectricFunction | dielectric function | DielectricFunction | exspy |
| EDS_SEM | | EDSEMSpectrum | exspy |
| EDS_TEM | | EDSTEMSpectrum | exspy |
| EELS | TEM EELS | EELSSpectrum | exspy |
| hologram | | HologramImage | holospy |
+-----+-----+-----+-----+
```

When *loading data*, the `signal_type` will be set automatically by the file reader, as defined in `rosettascio`. If the extension providing the corresponding signal subclass is installed, `load()` will return the subclass from the hyperspy extension, otherwise a warning will be raised to explain that no registered signal class can be assigned to the given `signal_type`.

Since the `load()` can return domain specific signal objects (e.g. `EDSSEMSpectrum` from `EleXSpy`) provided by extensions, the corresponding functionalities (so-called *method* of *object* in object-oriented programming, e.g. `EDSSEMSpectrum.get_lines_intensity()`) implemented in signal classes of the extension can be accessed directly. To use additional functionalities implemented in extensions, but not as method of the signal class, the extensions need to be imported explicitly (e.g. `import elexsy`). Check the user guides of the respective [HyperSpy extensions](#) for details on the provided methods and functions.

For details on how to write and register extensions see [Writing packages that extend HyperSpy](#).

4.1.5 Transforming between signal subclasses

The `BaseSignal` method `set_signal_type()` changes the `signal_type` in place, which may result in a `BaseSignal` subclass transformation.

The following example shows how to change the signal dimensionality and how to transform between different subclasses:

```
>>> s = hs.signals.Signal1D(np.random.random((10,20,100)))
>>> s
<Signal1D, title: , dimensions: (20, 10|100)>
>>> s.metadata
├── General
│   └── title =
└── Signal
    └── signal_type =
>>> im = s.to_signal2D()
>>> im
<Signal2D, title: , dimensions: (100|20, 10)>
>>> im.metadata
├── General
│   └── title =
└── Signal
    └── signal_type =
>>> s.set_signal_type("EELS")
>>> s
<EELSSpectrum, title: , dimensions: (20, 10|100)>
>>> s.metadata
├── General
│   └── title =
└── Signal
    └── signal_type = EELS
>>> s.change_dtype("complex")
>>> s
<ComplexSignal1D, title: , dimensions: (20, 10|100)>
```



```
>>> arr.shape
(2, 2)
```

In addition to the use of the keyword `ragged` when creating an hyperspy signal, the `ragged` attribute can also be set to specify whether the signal contains a ragged array or not.

In the following example, an hyperspy signal is created without specifying that the array is ragged. In this case, the signal dimension is 2, which *can be* misleading, because each item contains a list of numbers. To provide an unambiguous representation of the fact that the signal contains a ragged array, the `ragged` attribute can be set to `True`. By doing so, the signal space will be described as “ragged” and the navigation shape will become the same as the shape of the ragged array:

```
>>> arr = np.array([[1, 2, 3], [1]], dtype=object)
>>> s = hs.signals.BaseSignal(arr)
>>> s
<BaseSignal, title: , dimensions: (|2)>

>>> s.ragged = True
>>> s
<BaseSignal, title: , dimensions: (2|ragged)>
```

4.3 Binned and unbinned signals

Signals that are a histogram of a probability density function (pdf) should have the `is_binned` attribute of the signal axis set to `True`. The reason is that some methods operate differently on signals that are *binned*. An example of *binned* signals are EDS spectra, where the multichannel analyzer integrates the signal counts in every channel (=bin). Note that for 2D signals each signal axis has an `is_binned` attribute that can be set independently. For example, for the first signal axis: `signal.axes_manager.signal_axes[0].is_binned`.

The default value of the `is_binned` attribute is shown in the following table:

Table 2: Binned default values for the different subclasses.

BaseSignal subclass	binned	Library
<i>BaseSignal</i>	False	hyperspy
<i>Signal1D</i>	False	hyperspy
<code>exspy.signals.EELSSpectrum</code>	True	exSpy
<code>exspy.signals.EDSSEMSpectrum</code>	True	exSpy
<code>exspy.signals.EDSTEMSpectrum</code>	True	exSpy
<i>Signal2D</i>	False	hyperspy
<i>ComplexSignal</i>	False	hyperspy
<i>ComplexSignal1D</i>	False	hyperspy
<i>ComplexSignal2D</i>	False	hyperspy

To change the default value:

```
>>> s.axes_manager[-1].is_binned = True
```

Changed in version 1.7: The `binned` attribute from the metadata has been replaced by the axis attributes `is_binned`.

4.3.1 Integration of binned signals

For binned axes, the detector already provides the per-channel integration of the signal. Therefore, in this case, `integrate1D()` performs a simple summation along the given axis. In contrast, for unbinned axes, `integrate1D()` calls the `integrate_simpson()` method.

4.4 Indexing

Indexing a `BaseSignal` provides a powerful, convenient and Pythonic way to access and modify its data. In HyperSpy indexing is achieved using `isig` and `inav`, which allow the navigation and signal dimensions to be indexed independently. The idea is essentially to specify a subset of the data based on its position in the array and it is therefore essential to know the convention adopted for specifying that position, which is described here.

Those new to Python may find indexing a somewhat esoteric concept but once mastered it is one of the most powerful features of Python based code and greatly simplifies many common tasks. HyperSpy's Signal indexing is similar to numpy array indexing and those new to Python are encouraged to read the associated [numpy documentation on the subject](#).

Key features of indexing in HyperSpy are as follows (note that some of these features differ from numpy):

- HyperSpy indexing does:
 - Allow independent indexing of signal and navigation dimensions
 - Support indexing with decimal numbers.
 - Support indexing with units.
 - Support indexing with relative coordinates i.e. 'rel0.5'
 - Use the image order for indexing i.e. [x, y, z,...] (HyperSpy) vs [..., z, y, x] (numpy)
- HyperSpy indexing does not:
 - Support indexing using arrays.
 - Allow the addition of new axes using the `newaxis` object.

The examples below illustrate a range of common indexing tasks.

First consider indexing a single spectrum, which has only one signal dimension (and no navigation dimensions) so we use `isig`:

```
>>> s = hs.signals.Signal1D(np.arange(10))
>>> s
<Signal1D, title: , dimensions: (|10)>
>>> s.data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> s.isig[0]
<Signal1D, title: , dimensions: (|1)>
>>> s.isig[0].data
array([0])
>>> s.isig[9].data
array([9])
>>> s.isig[-1].data
array([9])
>>> s.isig[:5]
<Signal1D, title: , dimensions: (|5)>
```

(continues on next page)

(continued from previous page)

```

>>> s.isig[:5].data
array([0, 1, 2, 3, 4])
>>> s.isig[5::-1]
<Signal1D, title: , dimensions: (|6)>
>>> s.isig[5::-1]
<Signal1D, title: , dimensions: (|6)>
>>> s.isig[5::2]
<Signal1D, title: , dimensions: (|3)>
>>> s.isig[5::2].data
array([5, 7, 9])

```

Unlike numpy, HyperSpy supports indexing using decimal numbers or strings (containing a decimal number and units), in which case HyperSpy indexes using the axis scales instead of the indices. Additionally, one can index using relative coordinates, for example 'rel0.5' to index the middle of the axis.

```

>>> s = hs.signals.Signal1D(np.arange(10))
>>> s
<Signal1D, title: , dimensions: (|10)>
>>> s.data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> s.axes_manager[0].scale = 0.5
>>> s.axes_manager[0].axis
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
>>> s.isig[0.5:4].data
array([1, 2, 3, 4, 5, 6, 7])
>>> s.isig[0.5:4].data
array([1, 2, 3])
>>> s.isig[0.5:4:2].data
array([1, 3])
>>> s.axes_manager[0].units = 'µm'
>>> s.isig[:'2000 nm'].data
array([0, 1, 2, 3])
>>> s.isig[:'rel0.5'].data
array([0, 1, 2, 3])

```

Importantly the original *BaseSignal* and its “indexed self” share their data and, therefore, modifying the value of the data in one modifies the same value in the other. Note also that in the example below `s.data` is used to access the data as a numpy array directly and this array is then indexed using numpy indexing.

```

>>> s = hs.signals.Signal1D(np.arange(10))
>>> s
<Signal1D, title: , dimensions: (|10)>
>>> s.data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> si = s.isig[:,2]
>>> si.data
array([0, 2, 4, 6, 8])
>>> si.data[:] = 10
>>> si.data
array([10, 10, 10, 10, 10])
>>> s.data
array([10, 1, 10, 3, 10, 5, 10, 7, 10, 9])

```

(continues on next page)

(continued from previous page)

```
>>> s.data[:] = 0
>>> si.data
array([0, 0, 0, 0, 0])
```

Of course it is also possible to use the same syntax to index multidimensional data treating navigation axes using *inav* and signal axes using *isig*.

```
>>> s = hs.signals.Signal1D(np.arange(2*3*4).reshape((2,3,4)))
>>> s
<Signal1D, title: , dimensions: (3, 2|4)>
>>> s.data
array([[ [ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> s.axes_manager[0].name = 'x'
>>> s.axes_manager[1].name = 'y'
>>> s.axes_manager[2].name = 't'
>>> s.axes_manager.signal_axes
(<t axis, size: 4>,)
>>> s.axes_manager.navigation_axes
(<x axis, size: 3, index: 0>, <y axis, size: 2, index: 0>)
>>> s.inav[0,0].data
array([0, 1, 2, 3])
>>> s.inav[0,0].axes_manager
<Axes manager, axes: (|4)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
----- | ----- | ----- | ----- | ----- | -----
          t |      4 |      0 |      0 |      1 | <undefined>
>>> s.inav[0,0].isig[::-1].data
array([3, 2, 1, 0])
>>> s.isig[0]
<BaseSignal, title: , dimensions: (3, 2|)>
>>> s.isig[0].axes_manager
<Axes manager, axes: (3, 2|)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
          x |      3 |      0 |      0 |      1 | <undefined>
          y |      2 |      0 |      0 |      1 | <undefined>
----- | ----- | ----- | ----- | ----- | -----
>>> s.isig[0].data
array([[ 0,  4,  8],
       [12, 16, 20]])
```

Independent indexing of the signal and navigation dimensions is demonstrated further in the following:

```
>>> s = hs.signals.Signal1D(np.arange(2*3*4).reshape((2,3,4)))
>>> s
```

(continues on next page)

(continued from previous page)

```

<Signal1D, title: , dimensions: (3, 2|4)>
>>> s.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],

      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> s.axes_manager[0].name = 'x'
>>> s.axes_manager[1].name = 'y'
>>> s.axes_manager[2].name = 't'
>>> s.axes_manager.signal_axes
(<t axis, size: 4>,)
>>> s.axes_manager.navigation_axes
(<x axis, size: 3, index: 0>, <y axis, size: 2, index: 0>)
>>> s.inav[0,0].data
array([0, 1, 2, 3])
>>> s.inav[0,0].axes_manager
<Axes manager, axes: (|4)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
----- | ----- | ----- | ----- | ----- | -----
          t |     4 |     0 |     0 |     1 | <undefined>
>>> s.isig[0]
<BaseSignal, title: , dimensions: (3, 2|)>
>>> s.isig[0].axes_manager
<Axes manager, axes: (3, 2|)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
          x |     3 |     0 |     0 |     1 | <undefined>
          y |     2 |     0 |     0 |     1 | <undefined>
----- | ----- | ----- | ----- | ----- | -----
>>> s.isig[0].data
array([[ 0,  4,  8],
       [12, 16, 20]])

```

The same syntax can be used to set the data values in signal and navigation dimensions respectively:

```

>>> s = hs.signals.Signal1D(np.arange(2*3*4).reshape((2, 3, 4)))
>>> s
<Signal1D, title: , dimensions: (3, 2|4)>
>>> s.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],

      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> s.inav[0,0].data
array([0, 1, 2, 3])

```

(continues on next page)

(continued from previous page)

```

>>> s.inav[0,0] = 1
>>> s.inav[0,0].data
array([1, 1, 1, 1])
>>> s.inav[0,0] = s.inav[1,1]
>>> s.inav[0,0].data
array([16, 17, 18, 19])

```

4.5 Generic tools

Below we briefly introduce some of the most commonly used tools (methods). For more details about a particular method click on its name. For a detailed list of all the methods available see the [BaseSignal](#) documentation.

The methods of this section are available to all the signals. In other chapters methods that are only available in specialized subclasses are listed.

4.5.1 Mathematical operations

A number of mathematical operations are available in [BaseSignal](#). Most of them are just wrapped numpy functions.

The methods that perform mathematical operation over one or more axis at a time are:

- [sum\(\)](#)
- [max\(\)](#)
- [min\(\)](#)
- [mean\(\)](#)
- [std\(\)](#)
- [var\(\)](#)
- [nansum\(\)](#)
- [nanmax\(\)](#)
- [nanmin\(\)](#)
- [nanmean\(\)](#)
- [nanstd\(\)](#)
- [nanvar\(\)](#)

Note that by default all this methods perform the operation over *all* navigation axes.

Example:

```

>>> s = hs.signals.BaseSignal(np.random.random((2,4,6)))
>>> s.axes_manager[0].name = 'E'
>>> s
<BaseSignal, title: , dimensions: (|6, 4, 2)>
>>> # by default perform operation over all navigation axes
>>> s.sum()
<BaseSignal, title: , dimensions: (|6, 4, 2)>
>>> # can also pass axes individually

```

(continues on next page)

(continued from previous page)

```

>>> s.sum('E')
<Signal2D, title: , dimensions: (|4, 2)>
>>> # or a tuple of axes to operate on, with duplicates, by index or directly
>>> ans = s.sum((-1, s.axes_manager[1], 'E', 0))
>>> ans
<BaseSignal, title: , dimensions: (|1)>
>>> ans.axes_manager[0]
<Scalar axis, size: 1>

```

The following methods operate only on one axis at a time:

- `diff()`
- `derivative()`
- `integrate_simpson()`
- `integrate1D()`
- `indexmin()`
- `indexmax()`
- `valuemin()`
- `valuemax()`

All numpy ufunc can operate on *BaseSignal* instances, for example:

```

>>> s = hs.signals.Signal1D([0, 1])
>>> s.metadata.General.title = "A"
>>> s
<Signal1D, title: A, dimensions: (|2)>
>>> np.exp(s)
<Signal1D, title: exp(A), dimensions: (|2)>
>>> np.exp(s).data
array([1. , 2.71828183])
>>> np.power(s, 2)
<Signal1D, title: power(A, 2), dimensions: (|2)>
>>> np.add(s, s)
<Signal1D, title: add(A, A), dimensions: (|2)>
>>> np.add(hs.signals.Signal1D([0, 1]), hs.signals.Signal1D([0, 1]))
<Signal1D, title: add(Untitled Signal 1, Untitled Signal 2), dimensions: (|2)>

```

Notice that the title is automatically updated. When the signal has no title a new title is automatically generated:

```

>>> np.add(hs.signals.Signal1D([0, 1]), hs.signals.Signal1D([0, 1]))
<Signal1D, title: add(Untitled Signal 1, Untitled Signal 2), dimensions: (|2)>

```

Functions (other than unfucs) that operate on numpy arrays can also operate on *BaseSignal* instances, however they return a numpy array instead of a *BaseSignal* instance e.g.:

```

>>> np.angle(s)
array([0., 0.])

```

Note: For numerical **differentiation** and **integration**, use the proper methods `derivative()` and `integrate1D()`. In certain cases, particularly when operating on a non-uniform axis, the approximations using the `diff()` and `sum()`

methods will lead to erroneous results.

4.5.2 Signal operations

BaseSignal supports all the Python binary arithmetic operations (+, -, *, //, %, divmod(), pow(), **, <<, >>, &, ^, |), augmented binary assignments (+=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=), unary operations (-, +, abs() and ~) and rich comparisons operations (<, <=, ==, x!=y, <>, >, >=).

These operations are performed element-wise. When the dimensions of the signals are not equal [numpy broadcasting rules](#) apply independently for the navigation and signal axes.

Warning: Hyperspy does not check if the calibration of the signals matches.

In the following example *s2* has only one navigation axis while *s* has two. However, because the size of their first navigation axis is the same, their dimensions are compatible and *s2* is broadcasted to match *s*'s dimensions.

```
>>> s = hs.signals.Signal2D(np.ones((3,2,5,4)))
>>> s2 = hs.signals.Signal2D(np.ones((2,5,4)))
>>> s
<Signal2D, title: , dimensions: (2, 3|4, 5)>
>>> s2
<Signal2D, title: , dimensions: (2|4, 5)>
>>> s + s2
<Signal2D, title: , dimensions: (2, 3|4, 5)>
```

In the following example the dimensions are not compatible and an exception is raised.

```
>>> s = hs.signals.Signal2D(np.ones((3,2,5,4)))
>>> s2 = hs.signals.Signal2D(np.ones((3,5,4)))
>>> s
<Signal2D, title: , dimensions: (2, 3|4, 5)>
>>> s2
<Signal2D, title: , dimensions: (3|4, 5)>
>>> s + s2
Traceback (most recent call last):
  File "<ipython-input-55-044bb11a0bd9>", line 1, in <module>
    s + s2
  File "<string>", line 2, in __add__
  File "/home/fjd29/Python/hyperspy/hyperspy/signal.py", line 2686, in _binary_operator_
    ← ruler
    raise ValueError(exception_message)
ValueError: Invalid dimensions for this operation
```

Broadcasting operates exactly in the same way for the signal axes:

```
>>> s = hs.signals.Signal2D(np.ones((3,2,5,4)))
>>> s2 = hs.signals.Signal1D(np.ones((3, 2, 4)))
>>> s
<Signal2D, title: , dimensions: (2, 3|4, 5)>
>>> s2
<Signal1D, title: , dimensions: (2, 3|4)>
```

(continues on next page)

(continued from previous page)

```
>>> s + s2
<Signal2D, title: , dimensions: (2, 3|4, 5)>
```

In-place operators also support broadcasting, but only when broadcasting would not change the left most signal dimensions:

```
>>> s += s2
>>> s
<Signal2D, title: , dimensions: (2, 3|4, 5)>
>>> s2 += s
Traceback (most recent call last):
  File "<ipython-input-64-fdb9d3a69771>", line 1, in <module>
    s2 += s
  File "<string>", line 2, in __iadd__
  File "/home/fjd29/Python/hyperspy/hyperspy/signal.py", line 2737, in _binary_operator_
    ruler
    self.data = getattr(sdata, op_name)(odata)
ValueError: non-broadcastable output operand with shape (3,2,1,4) doesn't match the
broadcast shape (3,2,5,4)
```

4.5.3 Iterating over the navigation axes

BaseSignal instances are iterables over the navigation axes. For example, the following code creates a stack of 10 images and saves them in separate “png” files by iterating over the signal instance:

```
>>> image_stack = hs.signals.Signal2D(np.random.randint(10, size=(2, 5, 64,64)))
>>> for single_image in image_stack:
...     single_image.save("image %s.png" % str(image_stack.axes_manager.indices))
The "image (0, 0).png" file was created.
The "image (1, 0).png" file was created.
The "image (2, 0).png" file was created.
The "image (3, 0).png" file was created.
The "image (4, 0).png" file was created.
The "image (0, 1).png" file was created.
The "image (1, 1).png" file was created.
The "image (2, 1).png" file was created.
The "image (3, 1).png" file was created.
The "image (4, 1).png" file was created.
```

The data of the signal instance that is returned at each iteration is a view of the original data, a property that we can use to perform operations on the data. For example, the following code rotates the image at each coordinate by a given angle and uses the `stack()` function in combination with `list comprehensions` to make a horizontal “collage” of the image stack:

```
>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.datasets.ascent()*5]))
>>> image_stack.axes_manager[1].name = "x"
>>> image_stack.axes_manager[2].name = "y"
>>> for image, angle in zip(image_stack, (0, 45, 90, 135, 180)):
...     image.data[:] = scipy.ndimage.rotate(image.data, angle=angle,
...     reshape=False)
```

(continues on next page)

(continued from previous page)

```

>>> # clip data to integer range:
>>> image_stack.data = np.clip(image_stack.data, 0, 255)
>>> collage = hs.stack([image for image in image_stack], axis=0)
>>> collage.plot(scalebar=False)

```

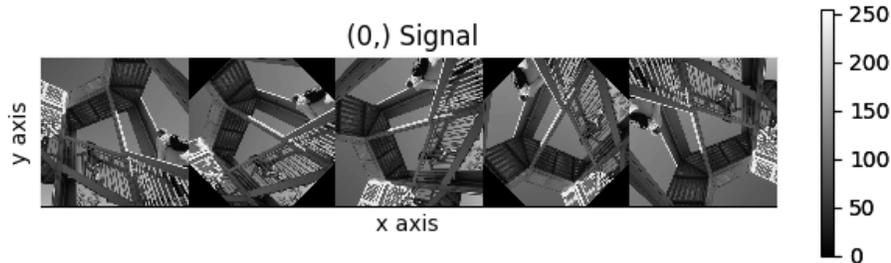


Fig. 2: Rotation of images by iteration.

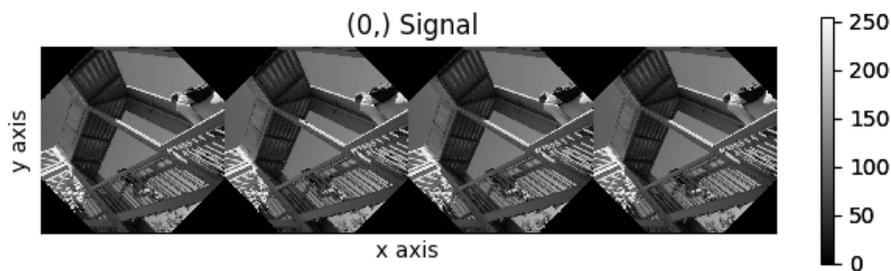
4.5.4 Iterating external functions with the map method

Performing an operation on the data at each coordinate, as in the previous example, using an external function can be more easily accomplished using the `map()` method:

```

>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.datasets.ascent()*4]))
>>> image_stack.axes_manager[1].name = "x"
>>> image_stack.axes_manager[2].name = "y"
>>> image_stack.map(scipy.ndimage.rotate, angle=45, reshape=False)
>>> # clip data to integer range
>>> image_stack.data = np.clip(image_stack.data, 0, 255)
>>> collage = hs.stack([image for image in image_stack], axis=0)
>>> collage.plot()

```

Fig. 3: Rotation of images by the same amount using `map()`.

The `map()` method can also take variable arguments as in the following example.

```

>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.datasets.ascent()*4]))
>>> image_stack.axes_manager[1].name = "x"

```

(continues on next page)

(continued from previous page)

```
>>> image_stack.axes_manager[2].name = "y"
>>> angles = hs.signals.BaseSignal(np.array([0, 45, 90, 135]))
>>> image_stack.map(scipy.ndimage.rotate, angle=angles.T, reshape=False)
```

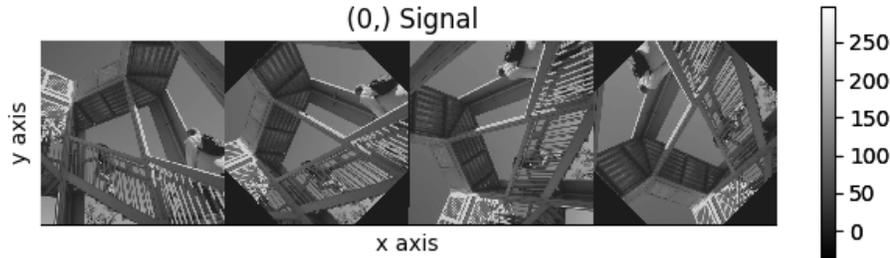


Fig. 4: Rotation of images using `map()` with different arguments for each image in the stack.

New in version 1.2.0: `inplace` keyword and non-preserved output shapes

If all function calls do not return identically-shaped results, only navigation information is preserved, and the final result is an array where each element corresponds to the result of the function (or arbitrary object type). These are *ragged arrays* and has the dtype *object*. As such, most HyperSpy functions cannot operate on such signals, and the data should be accessed directly.

The `inplace` keyword (by default `True`) of the `map()` method allows either overwriting the current data (default, `True`) or storing it to a new signal (`False`).

```
>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.datasets.ascent()*4])
>>> angles = hs.signals.BaseSignal(np.array([0, 45, 90, 135]))
>>> result = image_stack.map(scipy.ndimage.rotate,
...                          angle=angles.T,
...                          inplace=False,
...                          ragged=True,
...                          reshape=True)

>>> result
<BaseSignal, title: , dimensions: (4|ragged)>
>>> result.data.dtype
dtype('O')
>>> for d in result.data.flat:
...     print(d.shape)
(512, 512)
(724, 724)
(512, 512)
(724, 724)
```

New in version 1.4: Iterating over signal using a parameter with no navigation dimension.

In this case, the parameter is cyclically iterated over the navigation dimension of the input signal. In the example below, signal `s` is multiplied by a cosine parameter `d`, which is repeated over the navigation dimension of `s`.

```
>>> s = hs.signals.Signal1D(np.random.rand(10, 512))
>>> d = hs.signals.Signal1D(np.cos(np.linspace(0., 2*np.pi, 512)))
```

(continues on next page)

(continued from previous page)

```
>>> s.map(lambda A, B: A * B, B=d)
```

New in version 1.7: Get result as lazy signal

Especially when working with very large datasets, it can be useful to not do the computation immediately. For example if it would make you run out of memory. In that case, the *lazy_output* parameter can be used.

```
>>> from scipy.ndimage import gaussian_filter
>>> s = hs.signals.Signal2D(np.random.random((4, 4, 128, 128)))
>>> s_out = s.map(gaussian_filter, sigma=5, inplace=False, lazy_output=True)
>>> s_out
<LazySignal2D, title: , dimensions: (4, 4|128, 128)>
```

s_out can then be saved to a hard drive, to avoid it being loaded into memory. Alternatively, it can be computed and loaded into memory using *s_out.compute()*

```
>>> s_out.save("gaussian_filter_file.hspy")
```

Another advantage of using *lazy_output=True* is the ability to “chain” operations, by running *map()* on the output from a previous *map()* operation. For example, first running a Gaussian filter, followed by peak finding. This can improve the computation time, and reduce the memory need.

```
>>> s_out = s.map(scipy.ndimage.gaussian_filter, sigma=5, inplace=False, lazy_
↳output=True)
>>> from skimage.feature import blob_dog
>>> s_out1 = s_out.map(blob_dog, threshold=0.05, inplace=False, ragged=True, lazy_
↳output=False)
>>> s_out1
<BaseSignal, title: , dimensions: (4, 4|ragged)>
```

This is especially relevant for very large datasets, where memory use can be a limiting factor.

4.5.5 Cropping

Cropping can be performed in a very compact and powerful way using *Indexing*. In addition it can be performed using the following method or GUIs if cropping *signal1D* or *signal2D*. There is also a general *crop()* method that operates *in place*.

4.5.6 Rebinning

New in version 1.3: *rebin()* generalized to remove the constrain of the *new_shape* needing to be a divisor of *data.shape*.

The *rebin()* methods supports rebinning the data to arbitrary new shapes as long as the number of dimensions stays the same. However, internally, it uses two different algorithms to perform the task. Only when the new shape dimensions are divisors of the old shape’s, the operation supports *lazy-evaluation* and is usually faster. Otherwise, the operation requires linear interpolation.

For example, the following two equivalent rebinning operations can be performed lazily:

```
>>> s = hs.data.two_gaussians().as_lazy()
>>> print(s)
```

(continues on next page)

(continued from previous page)

```
<LazySignal1D, title: Two Gaussians, dimensions: (32, 32|1024)>
>>> print(s.rebin(scale=[1, 1, 2]))
<LazySignal1D, title: Two Gaussians, dimensions: (32, 32|512)>
```

```
>>> s = hs.data.two_gaussians().as_lazy()
>>> print(s.rebin(new_shape=[32, 32, 512]))
<LazySignal1D, title: Two Gaussians, dimensions: (32, 32|512)>
```

On the other hand, the following rebinning operation requires interpolation and cannot be performed lazily:

```
>>> s = hs.signals.Signal1D(np.ones([4, 4, 10]))
>>> s.data[1, 2, 9] = 5
>>> print(s)
<Signal1D, title: , dimensions: (4, 4|10)>
>>> print('Sum = ', s.data.sum())
Sum = 164.0
>>> scale = [0.5, 0.5, 5]
>>> test = s.rebin(scale=scale)
>>> test2 = s.rebin(new_shape=(8, 8, 2)) # Equivalent to the above
>>> print(test)
<Signal1D, title: , dimensions: (8, 8|2)>
>>> print(test2)
<Signal1D, title: , dimensions: (8, 8|2)>
>>> print('Sum =', test.data.sum())
Sum = 164.0
>>> print('Sum =', test2.data.sum())
Sum = 164.0
>>> s.as_lazy().rebin(scale=scale)
Traceback (most recent call last):
  File "<ipython-input-26-49bca19ebf34>", line 1, in <module>
    spectrum.as_lazy().rebin(scale=scale)
  File "/home/fjd29/Python/hyperspy3/hyperspy/_signals/eds.py", line 184, in rebin
    m = super().rebin(new_shape=new_shape, scale=scale, crop=crop, out=out)
  File "/home/fjd29/Python/hyperspy3/hyperspy/_signals/lazy.py", line 246, in rebin
    "Lazy rebin requires scale to be integer and divisor of the "
NotImplementedError: Lazy rebin requires scale to be integer and divisor of the original_
↪ signal shape
```

The dtype argument can be used to specify the dtype of the returned signal:

```
>>> s = hs.signals.Signal1D(np.ones((2, 5, 10), dtype=np.uint8))
>>> print(s)
<Signal1D, title: , dimensions: (5, 2|10)>
>>> print(s.data.dtype)
uint8
```

Use dtype=np.uint16 to specify a dtype:

```
>>> s2 = s.rebin(scale=(5, 2, 1), dtype=np.uint16)
>>> print(s2.data.dtype)
uint16
```

Use dtype="same" to keep the same dtype:

```
>>> s3 = s.rebin(scale=(5, 2, 1), dtype="same")
>>> print(s3.data.dtype)
uint8
```

By default `dtype=None`, the dtype is determined by the behaviour of `numpy.sum`, in this case, unsigned integer of the same precision as the platform integer:

```
>>> s4 = s.rebin(scale=(5, 2, 1))
>>> print(s4.data.dtype)
uint32
```

4.5.7 Interpolate to a different axis

The `interpolate_on_axis()` method makes it possible to exchange any existing axis of a signal with a new axis, regardless of the signals dimension or the axes types. This is achieved by interpolating the data using `scipy.interpolate.make_interp_spline()` from the old axis to the new axis. Replacing multiple axes can be done iteratively.

```
>>> from hyperspy.axes import UniformDataAxis, DataAxis
>>> x = {"offset": 0, "scale": 1, "size": 10, "name": "X", "navigate": True}
>>> e = {"offset": 0, "scale": 1, "size": 50, "name": "E", "navigate": False}
>>> s = hs.signals.Signal1D(np.random.random((10, 50)), axes=[x, e])
>>> s
<Signal1D, title: , dimensions: (10|50)>
>>> x_new = UniformDataAxis(offset=1.5, scale=0.8, size=7, name="X_NEW", navigate=True)
>>> e_new = DataAxis(axis=np.arange(8)**2, name="E_NEW", navigate=False)
>>> s2 = s.interpolate_on_axis(x_new, 0, inplace=False)
>>> s2
<Signal1D, title: , dimensions: (7|50)>
>>> s2.interpolate_on_axis(e_new, "E", inplace=True)
>>> s2
<Signal1D, title: , dimensions: (7|8)>
```

4.5.8 Squeezing

The `squeeze()` method removes any zero-dimensional axes, i.e. axes of `size=1`, and the attributed data dimensions from a signal. The method returns a reduced copy of the signal and does not operate in place.

```
>>> s = hs.signals.Signal2D(np.random.random((2, 1, 1, 6, 8, 8)))
>>> s
<Signal2D, title: , dimensions: (6, 1, 1, 2|8, 8)>
>>> s = s.squeeze()
>>> s
<Signal2D, title: , dimensions: (6, 2|8, 8)>
```

Squeezing can be particularly useful after a rebinning operation that leaves one dimension with `shape=1`:

```
>>> s = hs.signals.Signal2D(np.random.random((5, 5, 5, 10, 10)))
>>> s.rebin(new_shape=(5, 1, 5, 5, 5))
<Signal2D, title: , dimensions: (5, 1, 5|5, 5)>
```

(continues on next page)

(continued from previous page)

```
>>> s.rebin(new_shape=(5,1,5,5,5)).squeeze()
<Signal2D, title: , dimensions: (5, 5|5, 5)>
```

4.5.9 Folding and unfolding

When dealing with multidimensional datasets it is sometimes useful to transform the data into a two dimensional dataset. This can be accomplished using the following two methods:

- `fold()`
- `unfold()`

It is also possible to unfold only the navigation or only the signal space:

- `unfold_navigation_space()`
- `unfold_signal_space()`

4.5.10 Splitting and stacking

Several objects can be stacked together over an existing axis or over a new axis using the `stack()` function, if they share axis with same dimension.

```
>>> image = hs.signals.Signal2D(scipy.datasets.ascent())
>>> image = hs.stack([hs.stack([image]*3,axis=0)]*3,axis=1)
>>> image.plot()
```

Note: When stacking signals with large amount of `original_metadata`, these metadata will be stacked and this can lead to very large amount of metadata which can in turn slow down processing. The `stack_original_metadata` argument can be used to disable stacking `original_metadata`.

An object can be split into several objects with the `split()` method. This function can be used to reverse the `stack()` function:

```
>>> image = image.split()[0].split()[0]
>>> image.plot()
```

4.5.11 Fast Fourier Transform (FFT)

The `fast Fourier transform` of a signal can be computed using the `fft()` method. By default, the FFT is calculated with the origin at (0, 0), which will be displayed at the bottom left and not in the centre of the FFT. Conveniently, the `shift` argument of the `fft()` method can be used to center the output of the FFT. In the following example, the FFT of a hologram is computed using `shift=True` and its output signal is displayed, which shows that the FFT results in a complex signal with a real and an imaginary parts:

```
>>> im = hs.data.wave_image()
>>> fft_shifted = im.fft(shift=True)
>>> fft_shifted.plot()
```

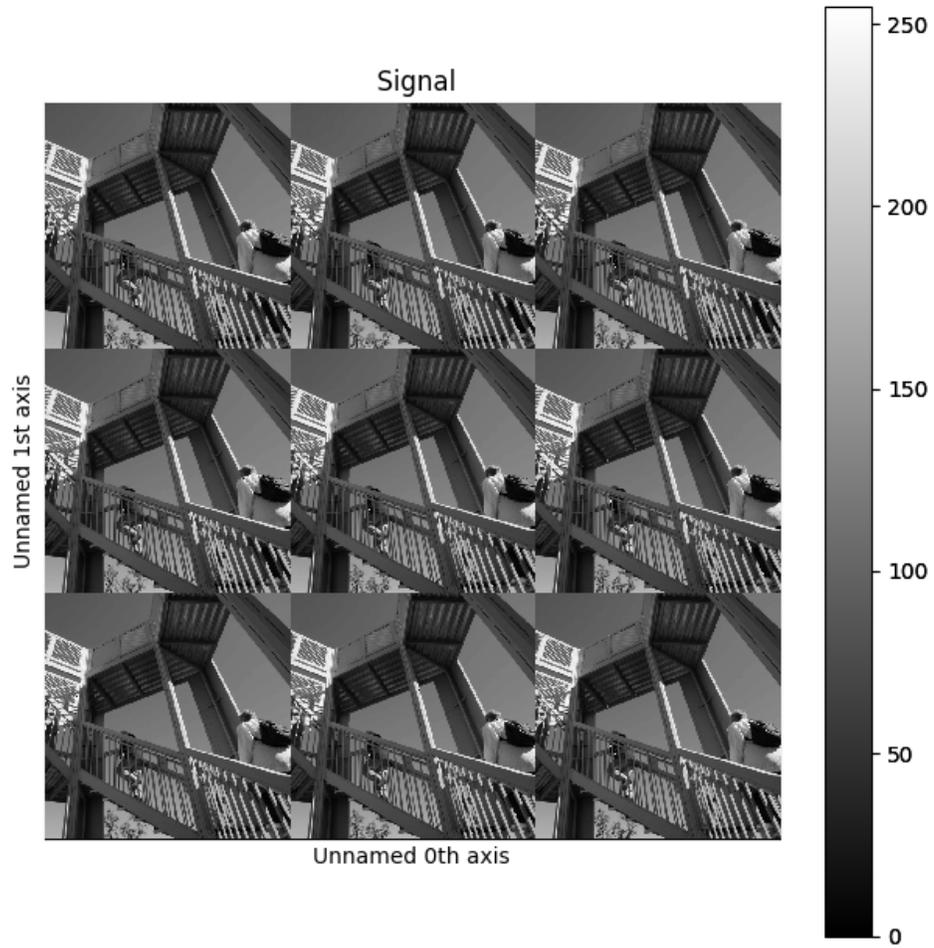


Fig. 5: Stacking example.

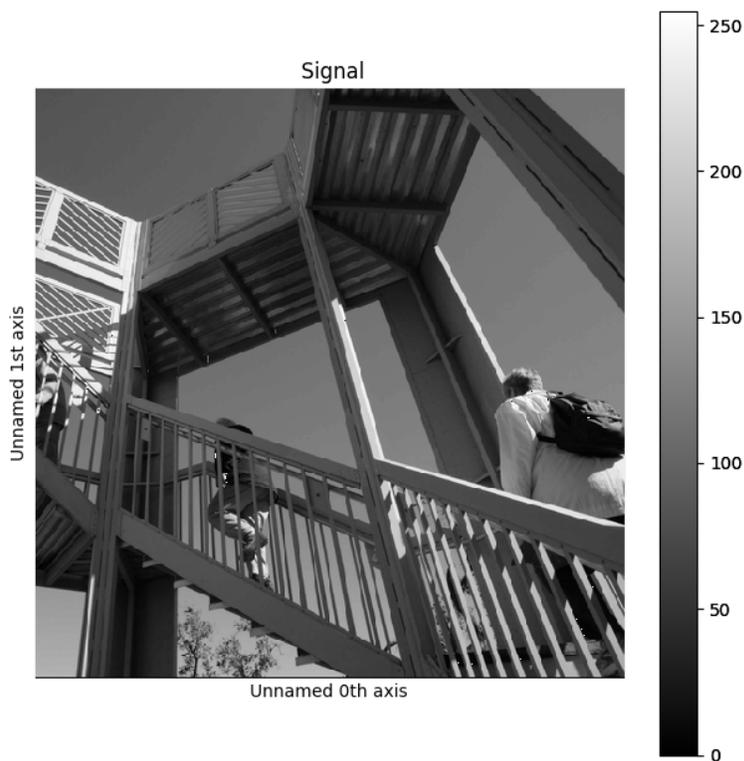
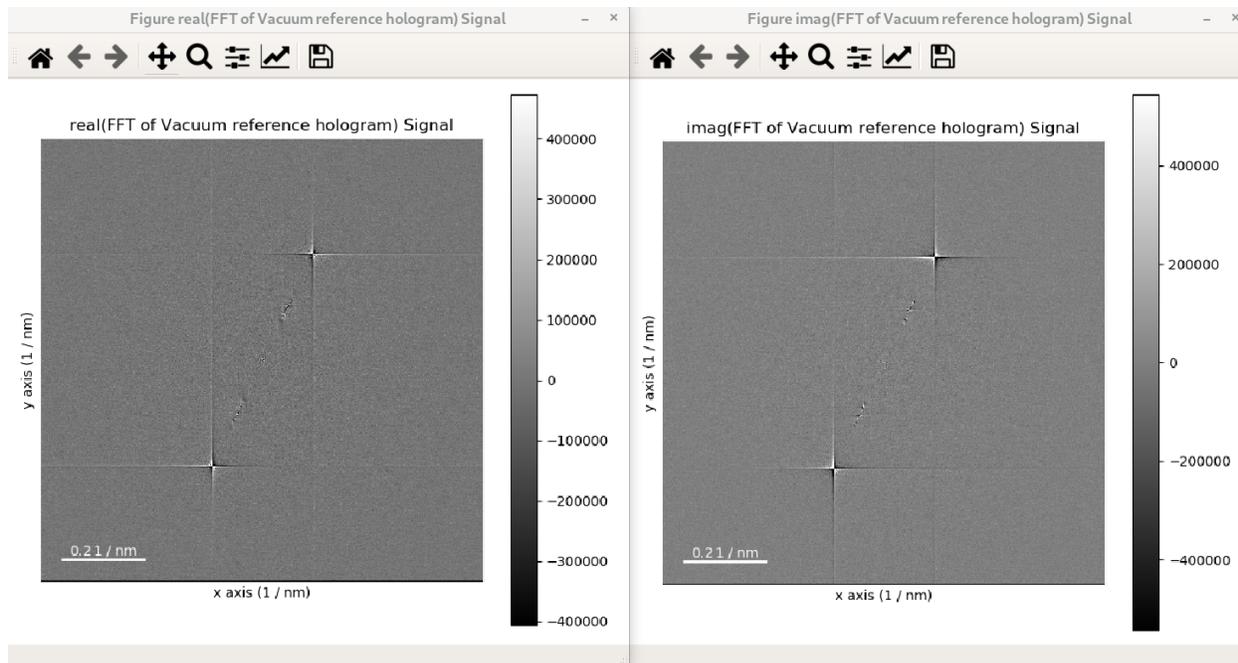


Fig. 6: Splitting example.

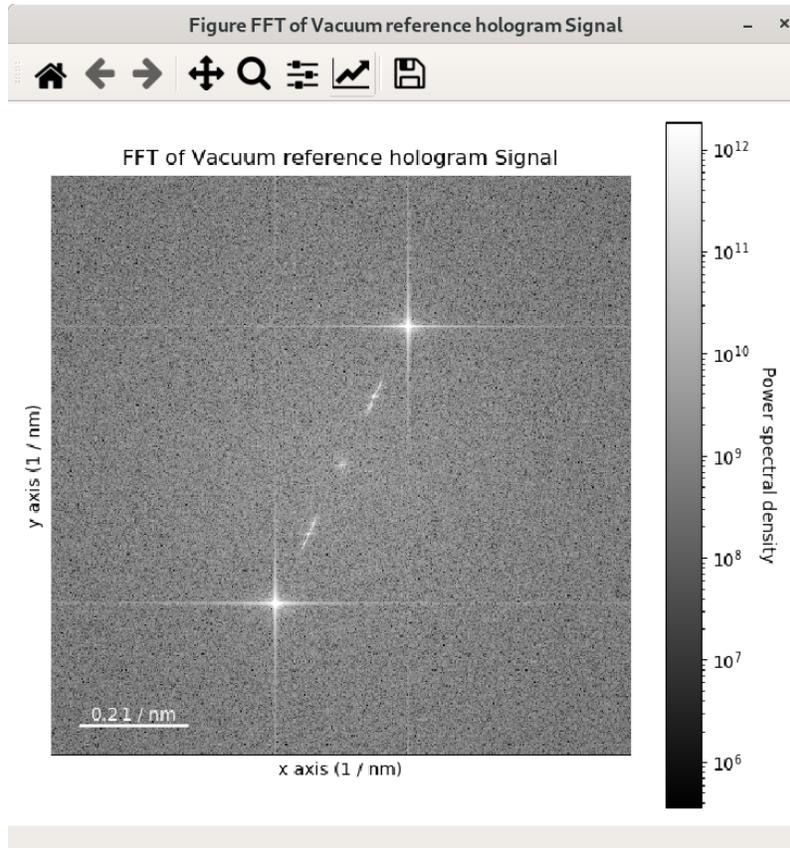


The strong features in the real and imaginary parts correspond to the lattice fringes of the hologram.

For visual inspection of the FFT it is convenient to display its power spectrum (i.e. the square of the absolute value of the FFT) rather than FFT itself as it is done in the example above by using the `power_spectrum` argument:

```
>>> im = hs.data.wave_image()
>>> fft = im.fft(True)
>>> fft.plot(True)
```

Where `power_spectrum` is set to `True` since it is the first argument of the `plot()` method for complex signal. When `power_spectrum=True`, the plot will be displayed on a log scale by default.



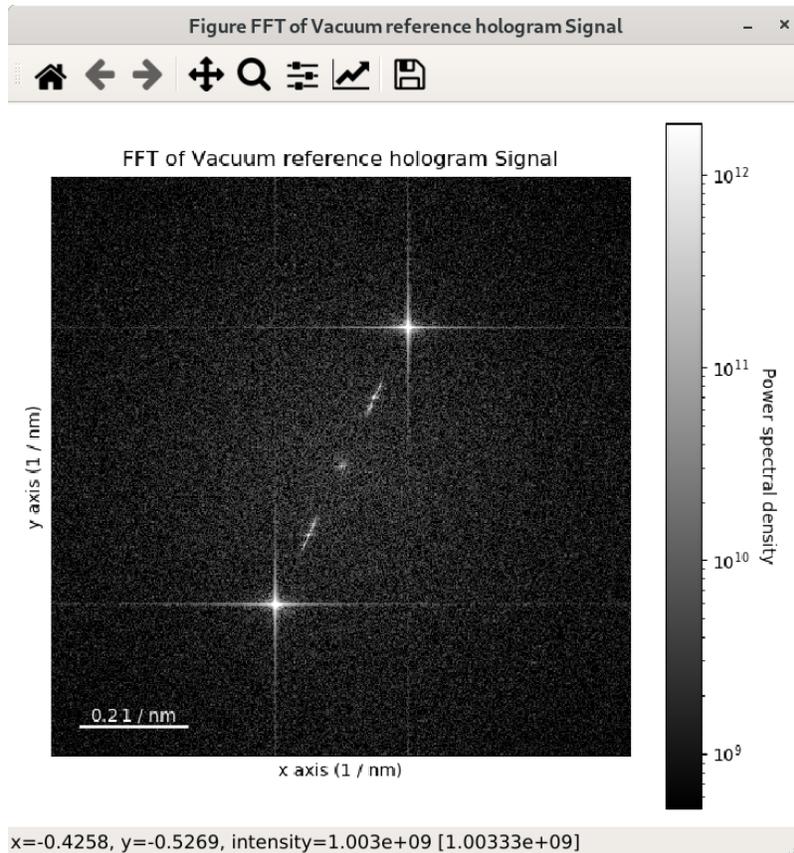
The visualisation can be further improved by setting the minimum value to display to the 30-th percentile; this can be done by using `vmin="30th"` in the plot function:

```
>>> im = hs.data.wave_image()
>>> fft = im.fft(True)
>>> fft.plot(True, vmin="30th")
```

The streaks visible in the FFT come from the edge of the image and can be removed by applying an [apodization](#) function to the original signal before the computation of the FFT. This can be done using the `apodization` argument of the `fft()` method and it is usually used for visualising FFT patterns rather than for quantitative analyses. By default, the so-called `hann` windows is used but different type of windows such as the `hamming` and `tukey` windows.

```
>>> im = hs.data.wave_image()
>>> fft = im.fft(shift=True)
```

(continues on next page)



(continued from previous page)

```
>>> fft_apodized = im.fft(shift=True, apodization=True)
>>> fft_apodized.plot(True, vmin="30th")
```

4.5.12 Inverse Fast Fourier Transform (iFFT)

Inverse fast Fourier transform can be calculated from a complex signal by using the `ifft()` method. Similarly to the `fft()` method, the `shift` argument can be provided to shift the origin of the iFFT when necessary:

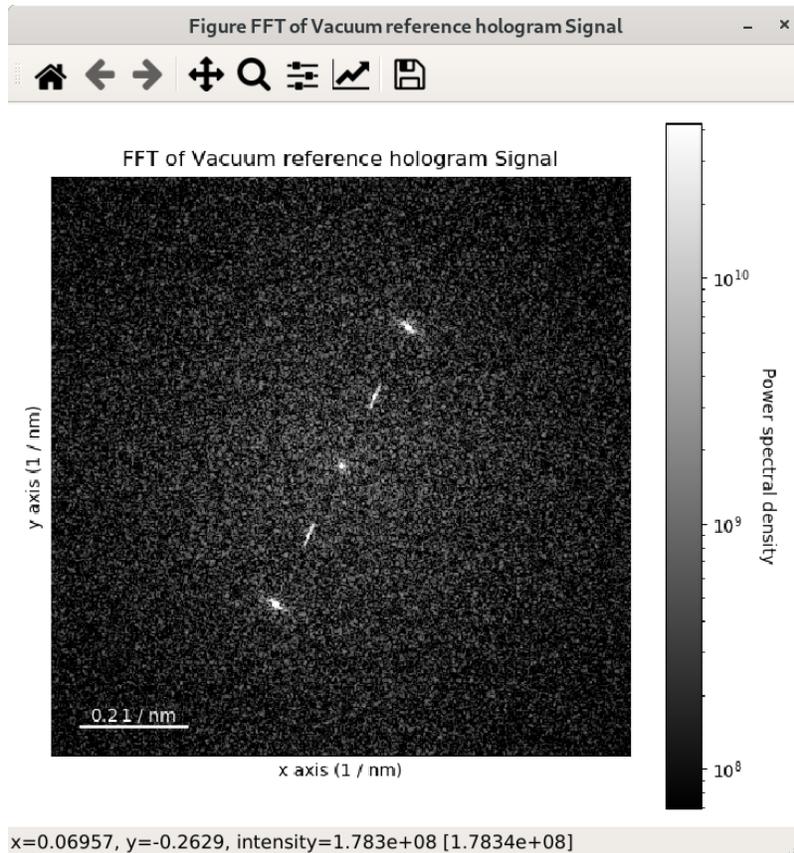
```
>>> im_ifft = im.fft(shift=True).ifft(shift=True)
```

4.5.13 Changing the data type

Even if the original data is recorded with a limited dynamic range, it is often desirable to perform the analysis operations with a higher precision. Conversely, if space is limited, storing in a shorter data type can decrease the file size. The `change_dtype()` changes the data type in place, e.g.:

```
>>> s = hs.load('EELS Signal1D Signal2D (high-loss).dm3')
Title: EELS Signal1D Signal2D (high-loss).dm3
Signal type: EELS
Data dimensions: (21, 42, 2048)
Data representation: spectrum
```

(continues on next page)



(continued from previous page)

```

Data type: float32
>>> s.change_dtype('float64')
>>> print(s)
Title: EELS Signal1D Signal2D (high-loss).dm3
Signal type: EELS
Data dimensions: (21, 42, 2048)
Data representation: spectrum
Data type: float64

```

In addition to all standard numpy dtypes, HyperSpy supports four extra dtypes for RGB images **for visualization purposes only**: `rgb8`, `rgba8`, `rgb16` and `rgba16`. This includes of course multi-dimensional RGB images.

The requirements for changing from and to any `rgbx` dtype are more strict than for most other dtype conversions. To change to a `rgbx` dtype the `signal_dimension` must be 1 and its size 3 (4) 3(4) for `rgb` (or `rgba`) dtypes and the dtype must be `uint8` (`uint16`) for `rgbx8` (`rgbx16`). After conversion the `signal_dimension` becomes 2.

Most operations on signals with RGB dtypes will fail. For processing simply change their dtype to `uint8` (`uint16`). The dtype of images of dtype `rgbx8` (`rgbx16`) can only be changed to `uint8` (`uint16`) and the `signal_dimension` becomes 1.

In the following example we create a 1D signal with signal size 3 and with dtype `uint16` and change its dtype to `rgb16` for plotting.

```

>>> rgb_test = np.zeros((1024, 1024, 3))
>>> ly, lx = rgb_test.shape[:2]

```

(continues on next page)

(continued from previous page)

```

>>> offset_factor = 0.16
>>> size_factor = 3
>>> Y, X = np.ogrid[0:lx, 0:ly]
>>> rgb_test[:, :, 0] = (X - lx / 2 - lx*offset_factor) ** 2 + \
...                     (Y - ly / 2 - ly*offset_factor) ** 2 < \
...                     lx * ly / size_factor **2
>>> rgb_test[:, :, 1] = (X - lx / 2 + lx*offset_factor) ** 2 + \
...                     (Y - ly / 2 - ly*offset_factor) ** 2 < \
...                     lx * ly / size_factor **2
>>> rgb_test[:, :, 2] = (X - lx / 2) ** 2 + \
...                     (Y - ly / 2 + ly*offset_factor) ** 2 \
...                     < lx * ly / size_factor **2
>>> rgb_test *= 2**16 - 1
>>> s = hs.signals.Signal1D(rgb_test)
>>> s.change_dtype("uint16")
>>> s
<Signal1D, title: , dimensions: (1024, 1024|3)>
>>> s.change_dtype("rgb16")
>>> s
<Signal2D, title: , dimensions: (|1024, 1024)>
>>> s.plot()

```

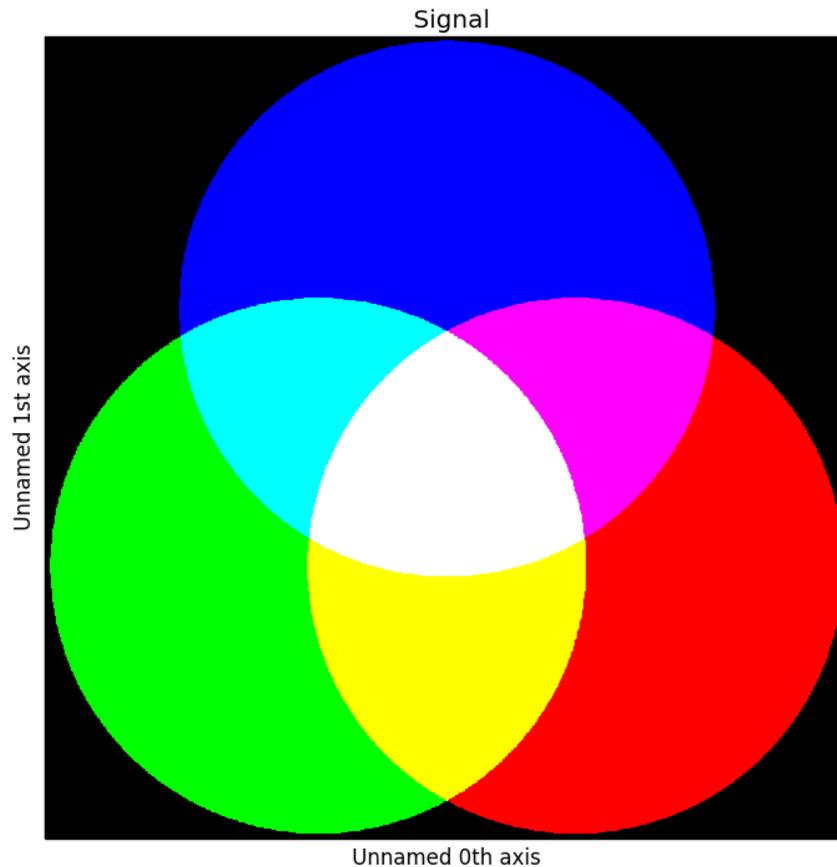


Fig. 7: RGB data type example.

4.5.14 Transposing (changing signal spaces)

New in version 1.1.

`transpose()` method changes how the dataset dimensions are interpreted (as signal or navigation axes). By default it swaps the signal and navigation axes. For example:

```
>>> s = hs.signals.Signal1D(np.zeros((4,5,6)))
>>> s
<Signal1D, title: , dimensions: (5, 4|6)>
>>> s.transpose()
<Signal2D, title: , dimensions: (6|5, 4)>
```

For `T()` is a shortcut for the default behaviour:

```
>>> s = hs.signals.Signal1D(np.zeros((4,5,6))).T
>>> s
<Signal2D, title: , dimensions: (6|5, 4)>
```

The method accepts both explicit axes to keep in either space, or just a number of axes required in one space (just one number can be specified, as the other is defined as “all other axes”). When axes order is not explicitly defined, they are “rolled” from one space to the other as if the `<navigation axes | signal axes >` wrap a circle. The example below should help clarifying this.

```
>>> # just create a signal with many distinct dimensions
>>> s = hs.signals.BaseSignal(np.random.rand(1, 2, 3, 4, 5, 6, 7, 8, 9))
>>> s
<BaseSignal, title: , dimensions: (|9, 8, 7, 6, 5, 4, 3, 2, 1)>
>>> s.transpose(signal_axes=5) # roll to leave 5 axes in signal space
<BaseSignal, title: , dimensions: (4, 3, 2, 1|9, 8, 7, 6, 5)>
>>> s.transpose(navigation_axes=3) # roll leave 3 axes in navigation space
<BaseSignal, title: , dimensions: (3, 2, 1|9, 8, 7, 6, 5, 4)>
>>> # 3 explicitly defined axes in signal space
>>> s.transpose(signal_axes=[0, 2, 6])
<BaseSignal, title: , dimensions: (8, 6, 5, 4, 2, 1|9, 7, 3)>
>>> # A mix of two lists, but specifying all axes explicitly
>>> # The order of axes is preserved in both lists
>>> s.transpose(navigation_axes=[1, 2, 3, 4, 5, 8], signal_axes=[0, 6, 7])
<BaseSignal, title: , dimensions: (8, 7, 6, 5, 4, 1|9, 3, 2)>
```

A convenience functions `transpose()` is available to operate on many signals at once, for example enabling plotting any-dimension signals trivially:

```
>>> s2 = hs.signals.BaseSignal(np.random.rand(2, 2)) # 2D signal
>>> s3 = hs.signals.BaseSignal(np.random.rand(3, 3, 3)) # 3D signal
>>> s4 = hs.signals.BaseSignal(np.random.rand(4, 4, 4, 4)) # 4D signal
>>> hs.plot.plot_images(hs.transpose(s2, s3, s4, signal_axes=2))
```

The `transpose()` method accepts keyword argument `optimize`, which is `False` by default, meaning modifying the output signal data **always modifies the original data** i.e. the data is just a view of the original data. If `True`, the method ensures the data in memory is stored in the most efficient manner for iterating by making a copy of the data if required, hence modifying the output signal data **not always modifies the original data**.

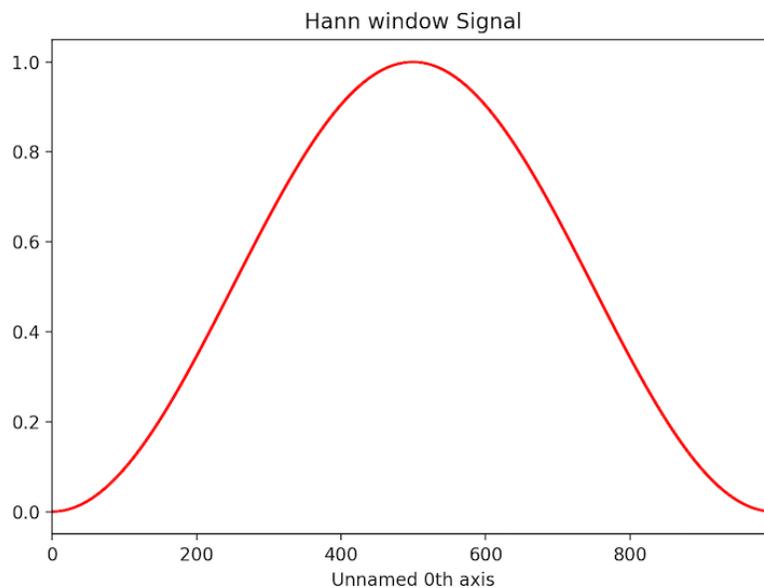
The convenience methods `as_signal1D()` and `as_signal2D()` internally use `transpose()`, but always optimize the data for iteration over the navigation axes if required. Hence, these methods do not always return a view of the original data. If a copy of the data is required use `deepcopy()` on the output of any of these methods e.g.:

```
>>> hs.signals.Signal1D(np.zeros((4,5,6))).T.deepcopy()
<Signal2D, title: , dimensions: (6|5, 4)>
```

4.5.15 Applying apodization window

Apodization window (also known as apodization function) can be applied to a signal using `apply_apodization()` method. By default standard Hann window is used:

```
>>> s = hs.signals.Signal1D(np.ones(1000))
>>> sa = s.apply_apodization()
>>> sa.metadata.General.title = 'Hann window'
>>> sa.plot()
```



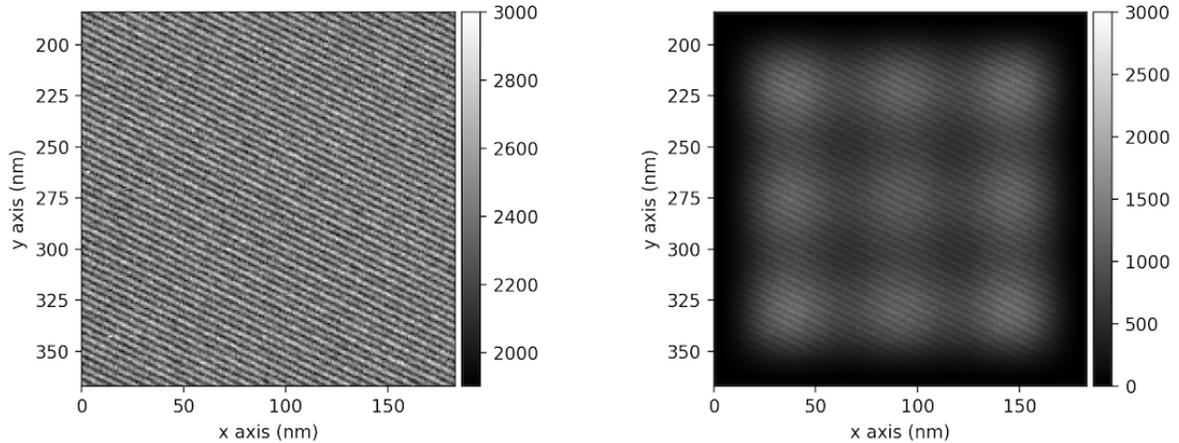
Higher order Hann window can be used in order to keep larger fraction of intensity of original signal. This can be done providing an integer number for the order of the window through keyword argument `hann_order`. (The last one works only together with default value of `window` argument or with `window='hann'`.)

```
>>> im = hs.data.wave_image().isig[:200, :200]
>>> ima = im.apply_apodization(window='hann', hann_order=3)
>>> hs.plot.plot_images([im, ima], vmax=3000, tight_layout=True)
[<Axes: >, <Axes: >]
```

In addition to Hann window also Hamming or Tukey windows can be applied using `window` attribute selecting `'hamming'` or `'tukey'` respectively.

The shape of Tukey window can be adjusted using parameter `alpha` provided through `tukey_alpha` keyword argument (only used when `window='tukey'`). The parameter represents the fraction of the window inside the cosine tapered region, i.e. smaller is `alpha` larger is the middle flat region where the original signal is preserved. If `alpha` is one, the Tukey window is equivalent to a Hann window. (Default value is 0.5)

Apodization can be applied in place by setting keyword argument `inplace` to `True`. In this case method will not return anything.



4.6 Basic statistical analysis

`get_histogram()` computes the histogram and conveniently returns it as signal instance. It provides methods to calculate the bins. `print_summary_statistics()` prints the five-number summary statistics of the data.

These two methods can be combined with `get_current_signal()` to compute the histogram or print the summary statistics of the signal at the current coordinates, e.g:

```
>>> s = hs.signals.Signal1D(np.random.normal(size=(10, 100)))
>>> s.print_summary_statistics()
Summary statistics
-----
mean:      -0.0143
std:       0.982
min:       -3.18
Q1:        -0.686
median:    0.00987
Q3:        0.653
max:       2.57

>>> s.get_current_signal().print_summary_statistics()
Summary statistics
-----
mean:      -0.019
std:       0.855
min:       -2.803
Q1:        -0.451
median:    -0.038
Q3:        0.484
max:       1.992
```

Histogram of different objects can be compared with the functions `plot_histograms()` (see [visualisation](#) for the plotting options). For example, with histograms of several random chi-square distributions:

```
>>> img = hs.signals.Signal2D([np.random.chisquare(i+1, [100, 100]) for
...                             i in range(5)])
>>> hs.plot.plot_histograms(img, legend='auto')
```

(continues on next page)

(continued from previous page)

```
<Axes: xlabel='value (<undefined>)', ylabel='Intensity'>
```

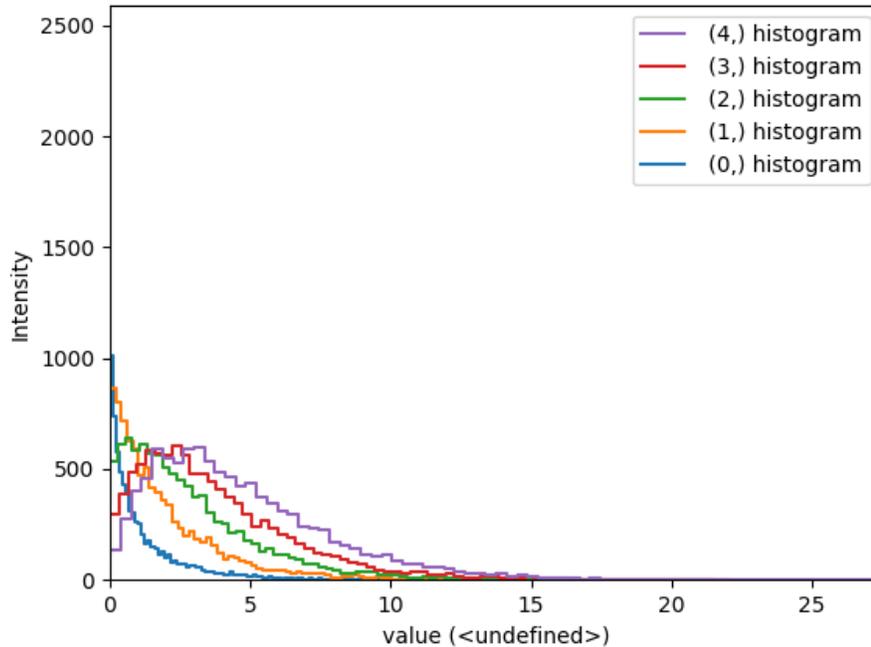


Fig. 8: Comparing histograms.

4.7 Setting the noise properties

Some data operations require the data variance. Those methods use the `metadata.Signal.Noise_properties.variance` attribute if it exists. You can set this attribute as in the following example where we set the variance to be 10:

```
>>> s.metadata.Signal.set_item("Noise_properties.variance", 10)
```

You can also use the functions `set_noise_variance()` and `get_noise_variance()` for convenience:

```
>>> s.set_noise_variance(10)
>>> s.get_noise_variance()
10
```

For heteroscedastic noise the `variance` attribute must be a `BaseSignal`. Poissonian noise is a common case of heteroscedastic noise where the variance is equal to the expected value. The `estimate_poissonian_noise_variance()` method can help setting the variance of data with semi-Poissonian noise. With the default arguments, this method simply sets the variance attribute to the given `expected_value`. However, more generally (although the noise is not strictly Poissonian), the variance may be proportional to the expected value. Moreover, when the noise is a mixture of white (Gaussian) and Poissonian noise, the variance is described by the following linear model:

$$\text{Var}[X] = (a * E[X] + b) * c$$

Where a is the `gain_factor`, b is the `gain_offset` (the Gaussian noise variance) and c the `correlation_factor`. The correlation factor accounts for correlation of adjacent signal elements that can be modelled as a convolution with a Gaussian point spread function. `estimate_poissonian_noise_variance()` can be used to set the noise properties when the variance can be described by this linear model, for example:

```
>>> s = hs.signals.Signal1D(np.ones(100))
>>> s.add_poissonian_noise()
>>> s.metadata
├── General
│   └── title =
├── Signal
│   └── signal_type =
└──
>>> s.estimate_poissonian_noise_variance()
>>> s.metadata
├── General
│   └── title =
├── Signal
│   ├── Noise_properties
│   │   ├── Variance_linear_model
│   │   │   ├── correlation_factor = 1
│   │   │   ├── gain_factor = 1
│   │   │   └── gain_offset = 0
│   │   └── variance = <BaseSignal, title: Variance of , dimensions: (|100)>
│   └── signal_type =
└──
```

4.8 Speeding up operations

4.8.1 Reusing a Signal for output

Many signal methods create and return a new signal. For fast operations, the new signal creation time is non-negligible. Also, when the operation is repeated many times, for example in a loop, the cumulative creation time can become significant. Therefore, many operations on `BaseSignal` accept an optional argument `out`. If an existing signal is passed to `out`, the function output will be placed into that signal, instead of being returned in a new signal. The following example shows how to use this feature to slice a `BaseSignal`. It is important to know that the `BaseSignal` instance passed in the `out` argument must be well-suited for the purpose. Often this means that it must have the same axes and data shape as the `BaseSignal` that would normally be returned by the operation.

```
>>> s = hs.signals.Signal1D(np.arange(10))
>>> s_sum = s.sum(0)
>>> s_sum.data
array([45])
>>> s.isig[:5].sum(0, out=s_sum)
>>> s_sum.data
array([10])
>>> s_roi = s.isig[:3]
>>> s_roi
```

(continues on next page)

(continued from previous page)

```
<Signal1D, title: , dimensions: (|3)>
>>> s.isig.__getitem__(slice(None, 5), out=s_roi)
>>> s_roi
<Signal1D, title: , dimensions: (|5)>
```

4.9 Complex datatype

The HyperSpy *ComplexSignal* signal class and its subclasses for 1-dimensional and 2-dimensional data allow the user to access complex properties like the `real` and `imag` parts of the data or the `amplitude` (also known as the modulus) and `phase` (also known as angle or argument) directly. Getting and setting those properties can be done as follows:

```
>>> s = hs.signals.ComplexSignal1D(np.arange(100) + 1j * np.arange(100))
>>> real = s.real # real is a new HS signal accessing the same data
>>> s.real = np.random.random(100) # new_real can be an array or signal
>>> imag = s.imag # imag is a new HS signal accessing the same data
>>> s.imag = np.random.random(100) # new_imag can be an array or signal
```

It is important to note that *data* passed to the constructor of a *ComplexSignal* (or to a subclass), which is not already complex, will be converted to the numpy standard of `np.complex/np.complex128`. *data* which is already complex will be passed as is.

To transform a real signal into a complex one use:

```
>>> s.change_dtype(complex)
```

Changing the dtype of a complex signal to something real is not clearly defined and thus not directly possible. Use the `real`, `imag`, `amplitude` or `phase` properties instead to extract the real data that is desired.

4.9.1 Calculate the angle / phase / argument

The `angle()` function can be used to calculate the angle, which is equivalent to using the `phase` property if no argument is used. If the data is real, the angle will be 0 for positive values and 2π for negative values. If the `deg` parameter is set to `True`, the result will be given in degrees, otherwise in rad (default). The underlying function is the `numpy.angle()` function. `angle()` will return an appropriate HyperSpy signal.

4.9.2 Phase unwrapping

With the `unwrapped_phase()` method the complex phase of a signal can be unwrapped and returned as a new signal. The underlying method is `skimage.restoration.unwrap_phase()`, which uses the algorithm described in [Herraez].

4.9.3 Calculate and display Argand diagram

Sometimes it is convenient to visualize a complex signal as a plot of its imaginary part versus real one. In this case so called Argand diagrams can be calculated using `argand_diagram()` method, which returns the plot as a `Signal2D`. Optional arguments `size` and `display_range` can be used to change the size (and therefore resolution) of the plot and to change the range for the display of the plot respectively. The last one is especially useful in order to zoom into specific regions of the plot or to limit the plot in case of noisy data points.

An example of calculation of Argand diagram is `holospy:ref:shown for electron holography data <holo.argand-example>`.

4.9.4 Add a linear phase ramp

For 2-dimensional complex images, a linear phase ramp can be added to the signal via the `add_phase_ramp()` method. The parameters `ramp_x` and `ramp_y` dictate the slope of the ramp in x - and y direction, while the offset is determined by the `offset` parameter. The fulcrum of the linear ramp is at the origin and the slopes are given in units of the axis with the according scale taken into account. Both are available via the `AxesManager` of the signal.

4.10 GPU support

New in version 1.7.

GPU processing is supported thanks to the numpy dispatch mechanism of array functions - read [NEP-18](#) and [NEP-35](#) for more information. It means that most HyperSpy functions will work on a GPU if the data is a `cupy.ndarray` and the required functions are implemented in `cupy`.

Note: GPU processing with hyperspy requires `numpy>=1.20` and `dask>=2021.3.0`, to be able to use [NEP-18](#) and [NEP-35](#).

```
>>> import cupy as cp
>>> # Create a cupy array (on GPU device)
>>> data = cp.random.random(size=(20, 20, 100, 100))
>>> s = hs.signals.Signal2D(data)
>>> type(s.data)
... cupy._core.core.ndarray
```

Two convenience methods are available to transfer data between the host and the (GPU) device memory:

- `to_host()`
- `to_device()`

For lazy processing, see the *corresponding section*.

AXES HANDLING

5.1 The navigation and signal dimensions

HyperSpy distinguishes between *signal* and *navigation* axes and most functions operate on the *signal* axes and iterate over the *navigation* axes. Take an EELS spectrum image as specific example. It is a 2D array of spectra and has three dimensions: X, Y and energy-loss. In HyperSpy, X and Y are the *navigation* dimensions and the energy-loss is the *signal* dimension. To make this distinction more explicit, the representation of the object includes a separator | between the navigation and signal dimensions. In analogy, the *signal* dimension in EDX would be the X-ray energy, in optical spectra the wavelength axis, etc. However, HyperSpy can also handle data with more than one *signal* dimension, such as a stack or even map of diffraction images or electron-holograms in TEM.

For example: A spectrum image has signal dimension 1 and navigation dimension 2 and is stored in the Signal1D subclass.

```
>>> s = hs.signals.Signal1D(np.zeros((10, 20, 30)))
>>> s
<Signal1D, title: , dimensions: (20, 10|30)>
```

An image stack has signal dimension 2 and navigation dimension 1 and is stored in the Signal2D subclass.

```
>>> im = hs.signals.Signal2D(np.zeros((30, 10, 20)))
>>> im
<Signal2D, title: , dimensions: (30|20, 10)>
```

A map of images has signal dimension 2 and navigation dimension 2 and is stored in the Signal2D subclass.

```
>>> im = hs.signals.Signal2D(np.zeros((30, 10, 10, 20)))
>>> im
<Signal2D, title: , dimensions: (10, 30|20, 10)>
```

5.2 Setting axis properties

The axes are managed and stored by the [AxesManager](#) class that is stored in the `axes_manager` attribute of the signal class. The individual axes can be accessed by indexing the [AxesManager](#), e.g.:

```
>>> s = hs.signals.Signal1D(np.random.random((10, 20, 100)))
>>> s
<Signal1D, title: , dimensions: (20, 10|100)>
>>> s.axes_manager
```

(continues on next page)

(continued from previous page)

```
<Axes manager, axes: (20, 10|100)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
<undefined> | 20 | 0 | 0 | 1 | <undefined>
<undefined> | 10 | 0 | 0 | 1 | <undefined>
----- | ----- | ----- | ----- | ----- | -----
<undefined> | 100 | 0 | 0 | 1 | <undefined>
>>> s.axes_manager[0]
<Unnamed 0th axis, size: 20, index: 0>
```

The navigation axes come first, followed by the signal axes. Alternatively, it is possible to selectively access the navigation or signal dimensions:

```
>>> s.axes_manager.navigation_axes[1]
<Unnamed 1st axis, size: 10, index: 0>
>>> s.axes_manager.signal_axes[0]
<Unnamed 2nd axis, size: 100>
```

For the given example of two navigation and one signal dimensions, all the following commands will access the same axis:

```
>>> s.axes_manager[2]
<Unnamed 2nd axis, size: 100>
>>> s.axes_manager[-1]
<Unnamed 2nd axis, size: 100>
>>> s.axes_manager.signal_axes[0]
<Unnamed 2nd axis, size: 100>
```

The axis properties can be set by setting the *BaseDataAxis* attributes, e.g.:

```
>>> s.axes_manager[0].name = "X"
>>> s.axes_manager[0]
<X axis, size: 20, index: 0>
```

Once the name of an axis has been defined it is possible to request it by its name e.g.:

```
>>> s.axes_manager["X"]
<X axis, size: 20, index: 0>
>>> s.axes_manager["X"].scale = 0.2
>>> s.axes_manager["X"].units = "nm"
>>> s.axes_manager["X"].offset = 100
```

It is also possible to set the axes properties using a GUI by calling the *gui()* method of the *AxesManager*

```
>>> s.axes_manager.gui()
```

or, for a specific axis, the respective method of e.g. *UniformDataAxis*:

```
>>> s.axes_manager["X"].gui()
```

To simply change the “current position” (i.e. the indices of the navigation axes) you could use the navigation sliders:

```
>>> s.axes_manager.gui_navigation_sliders()
```

The screenshot shows two panels for configuring axes. The left panel is for 'Axis 0' and the right panel is for 'Axis 2'. Both panels have a close button (x) in the top left corner.

Axis 0 configuration:

- Name: X
- Size: 20
- Index in array: 1
- Index: 0 (slider)
- Value: 100.00 (slider)
- Units: nm
- Scale: 0.2
- Offset: 100

Axis 2 configuration:

- Name: (empty)
- Size: 100
- Index in array: 2
- Units: (empty)
- Scale: 1
- Offset: 0

At the bottom of the left panel, there is a button labeled 'Axis 1'.

Fig. 1: AxesManager ipywidgets GUI.

The screenshot shows the configuration for a single axis named 'X'. It has a close button (x) in the top left corner.

- Name: X
- Size: 20
- Index in array: 1
- Index: 0 (slider)
- Value: 100.00 (slider)
- Units: nm
- Scale: 0.2
- Offset: 100

Fig. 2: UniformDataAxis ipywidgets GUI.

The screenshot shows two sliders and a checkbox. The first slider is labeled 'X' and has a value of 0. The second slider is labeled 'Unnamed..' and also has a value of 0. To the right of the sliders, there are two input fields: the top one contains '100' and is followed by 'nm', and the bottom one contains '0'. A checkbox labeled 'Continous update' is checked.

Fig. 3: Navigation sliders ipywidgets GUI.

Alternatively, the “current position” can be changed programmatically by directly accessing the `indices` attribute of a signal’s `AxesManager` or the `index` attribute of an individual axis. This is particularly useful when trying to set a specific location at which to initialize a model’s parameters to sensible values before performing a fit over an entire spectrum image. The `indices` must be provided as a tuple, with the same length as the number of navigation dimensions:

```
>>> s.axes_manager.indices = (5, 4)
```

5.2.1 Summary of axis properties

- `name` (str) and `units` (str) are basic parameters describing an axis used in plotting. The latter enables the *conversion of units*.
- `navigate` (bool) determines, whether it is a navigation axis.
- `size` (int) gives the number of elements in an axis.
- `index` (int) determines the “current position for a navigation axis and `value` (float) returns the value at this position.
- `low_index` (int) and `high_index` (int) are the first and last index.
- `low_value` (int) and `high_value` (int) are the smallest and largest value.
- The axis array stores the values of the axis points. However, depending on the type of axis, this array may be updated from the **defining attributes** as discussed in the following section.

5.3 Types of data axes

HyperSpy supports different *data axis types*, which differ in how the axis is defined:

- `DataAxis` defined by an array axis,
- `FunctionalDataAxis` defined by a function expression or
- `UniformDataAxis` defined by the initial value offset and spacing scale.

The main disambiguation is whether the axis is **uniform**, where the data points are equidistantly spaced, or **non-uniform**, where the spacing may vary. The latter can become important when, e.g., a spectrum recorded over a *wavelength* axis is converted to a *wavenumber* or *energy* scale, where the conversion is based on a $1/x$ dependence so that the axis spacing of the new axis varies along the length of the axis. Whether an axis is uniform or not can be queried through the property `is_uniform` (bool) of the axis.

Every axis of a signal object may be of a different type. For example, it is common that the *navigation* axes would be *uniform*, while the *signal* axes are *non-uniform*.

When an axis is created, the type is automatically determined by the attributes passed to the generator. The three different axis types are summarized in the following table.

Table 1: BaseDataAxis subclasses.

BaseDataAxis subclass	Defining attributes	is_uniform
<code>DataAxis</code>	axis	False
<code>FunctionalDataAxis</code>	expression	False
<code>UniformDataAxis</code>	offset, scale	True

Note: Not all features are implemented for non-uniform axes.

Warning: Non-uniform axes are in beta state and its API may change in a minor release. Not all hyperspy features are compatible with non-uniform axes and support will be added in future releases.

5.3.1 Uniform data axis

The most common case is the *UniformDataAxis*. Here, the axis is defined by the *offset*, *scale* and *size* parameters, which determine the *initial value*, *spacing* and *length*, respectively. The actual axis array is automatically calculated from these three values. The *UniformDataAxis* is a special case of the *FunctionalDataAxis* defined by the function $\text{scale} * x + \text{offset}$.

Sample dictionary for a *UniformDataAxis*:

```
>>> dict0 = {'offset': 300, 'scale': 1, 'size': 500}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'UniformDataAxis', 'name': None, 'units': None, 'navigate': False, 'is_binned':
  False, 'size': 500, 'scale': 1.0, 'offset': 300.0}
```

Corresponding output of *AxesManager*:

```
>>> s.axes_manager
<Axes manager, axes: (|500)>
      Name | size | index | offset | scale | units
-----|-----|-----|-----|-----|-----
-----|-----|-----|-----|-----|-----
<undefined> | 500 | 0 | 3e+02 | 1 | <undefined>
```

5.3.2 Functional data axis

Alternatively, a *FunctionalDataAxis* is defined based on an expression that is evaluated to yield the axis points. The *expression* is a function defined as a string using the *SymPy* text expression format. An example would be $\text{expression} = a / x + b$. Any variables in the expression, in this case *a* and *b* must be defined as additional attributes of the axis. The property *is_uniform* is automatically set to *False*.

x itself is an instance of *BaseDataAxis*. By default, it will be a *UniformDataAxis* with *offset* = 0 and *scale* = 1 of the given *size*. However, it can also be initialized with custom *offset* and *scale* values. Alternatively, it can be a non uniform *DataAxis*.

Sample dictionary for a *FunctionalDataAxis*:

```
>>> dict0 = {'expression': 'a / (x + 1) + b', 'a': 100, 'b': 10, 'size': 500}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'FunctionalDataAxis', 'name': None, 'units': None, 'navigate': False, 'is_
  binned': False, 'expression': 'a / (x + 1) + b', 'size': 500, 'x': {'_type':
  'UniformDataAxis', 'name': None, 'units': None, 'navigate': False, 'is_binned': False,
  'size': 500, 'scale': 1.0, 'offset': 0.0}, 'a': 100, 'b': 10}
```

Corresponding output of *AxesManager*:

```
>>> s.axes_manager
<Axes manager, axes: (|500)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
----- | ----- | ----- | ----- | ----- | -----
<undefined> | 500 | 0 | non-uniform axis | <undefined>
```

Initializing *x* with offset and scale:

```
>>> from hyperspy.axes import UniformDataAxis
>>> dict0 = {'expression': 'a / x + b', 'a': 100, 'b': 10, 'x': UniformDataAxis(size=10,
↳ offset=10, scale=0.1)}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> # the x array
>>> s.axes_manager[0].x.axis
array([10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9])
>>> # the actual axis array
>>> s.axes_manager[0].axis
array([20.          , 19.9009901 , 19.80392157, 19.70873786, 19.61538462,
       19.52380952, 19.43396226, 19.34579439, 19.25925926, 19.17431193])
```

Initializing *x* as non-uniform *DataAxis*:

```
>>> from hyperspy.axes import DataAxis
>>> dict0 = {'expression': 'a / x + b', 'a': 100, 'b': 10, 'x': DataAxis(axis=np.
↳ arange(1,10)**2)}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> # the x array
>>> s.axes_manager[0].x.axis
array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
>>> # the actual axis array
>>> s.axes_manager[0].axis
array([110.          ,  35.          ,  21.11111111,  16.25          ,
       14.          ,  12.77777778,  12.04081633,  11.5625          ,
       11.2345679  ])
```

Initializing *x* with offset and scale:

5.3.3 (non-uniform) Data axis

A *DataAxis* is the most flexible type of axis. The axis points are directly given by an array named *axis*. As this can be any array, the property *is_uniform* is automatically set to *False*.

Sample dictionary for a *DataAxis*:

```
>>> dict0 = {'axis': np.arange(12)**2}
>>> s = hs.signals.Signal1D(np.ones(12), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'DataAxis', 'name': None, 'units': None, 'navigate': False, 'is_binned': False,
↳ 'axis': array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81, 100, 121])}
```

Corresponding output of *AxesManager*:

```
>>> s.axes_manager
<Axes manager, axes: (|12)>
      Name | size | index | offset | scale | units
-----|-----|-----|-----|-----|-----
-----|-----|-----|-----|-----|-----
<undefined> | 12 | 0 | non-uniform axis | <undefined>
```

5.4 Defining a new axis

An axis object can be created through the `axes.create_axis()` method, which automatically determines the type of axis by the given attributes:

```
>>> from hyperspy import axes
>>> axis = axes.create_axis(offset=10, scale=0.5, size=20)
>>> axis
<Unnamed axis, size: 20>
```

Alternatively, the creator of the different types of axes can be called directly:

```
>>> from hyperspy import axes
>>> axis = axes.UniformDataAxis(offset=10, scale=0.5, size=20)
>>> axis
<Unnamed axis, size: 20>
```

The dictionary defining the axis is returned by the `get_axis_dictionary()` method:

```
>>> axis.get_axis_dictionary()
{'_type': 'UniformDataAxis', 'name': None, 'units': None, 'navigate': False, 'is_binned':
↳ False, 'size': 20, 'scale': 0.5, 'offset': 10.0}
```

This dictionary can be used, for example, in the *initialization of a new signal*.

5.4.1 Adding/Removing axes to/from a signal

Usually, the axes are directly added to a signal during *signal initialization*. However, you may wish to add/remove axes from the *AxesManager* of a signal.

Note that there is currently no consistency check whether a signal object has the right number of axes of the right dimensions. Most functions will however fail if you pass a signal object where the axes do not match the data dimensions and shape.

You can *add a set of axes* to the *AxesManager* by passing either a list of axes dictionaries to `axes_manager.create_axes()`:

```
>>> dict0 = {'offset': 300, 'scale': 1, 'size': 500}
>>> dict1 = {'axis': np.arange(12)**2}
>>> s.axes_manager.create_axes([dict0, dict1])
```

or a list of axes objects:

```
>>> from hyperspy.axes import UniformDataAxis, DataAxis
>>> axis0 = UniformDataAxis(offset=300, scale=1, size=500)
>>> axis1 = DataAxis(axis=np.arange(12)**2)
>>> s.axes_manager.create_axes([axis0, axis1])
```

Remove an axis from the *AxesManager* using `remove()`, e.g. for the last axis:

```
>>> s.axes_manager.remove(-1)
```

5.5 Using quantity and converting units

The scale and the offset of each *UniformDataAxis* axis can be set and retrieved as quantity.

```
>>> s = hs.signals.Signal1D(np.arange(10))
>>> s.axes_manager[0].scale_as_quantity
<Quantity(1.0, 'dimensionless')>
>>> s.axes_manager[0].scale_as_quantity = '2.5 μm'
>>> s.axes_manager
<Axes manager, axes: (|10)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
----- | ----- | ----- | ----- | ----- | -----
<undefined> | 10 | 0 | 0 | 2.5 | μm
>>> s.axes_manager[0].offset_as_quantity = '2.5 nm'
```

Internally, HyperSpy uses the `pint` library to manage the scale and offset quantities. The `scale_as_quantity` and `offset_as_quantity` attributes return pint object:

```
>>> q = s.axes_manager[0].offset_as_quantity
>>> type(q) # q is a pint quantity object
<class 'pint.Quantity'>
>>> q
<Quantity(2.5, 'nanometer')>
```

The `convert_units` method of the *AxesManager* converts units, which by default (no parameters provided) converts all axis units to an optimal unit to avoid using too large or small numbers.

Each axis can also be converted individually using the `convert_to_units` method of the *UniformDataAxis*:

```
>>> axis = hs.hyperspy.axes.UniformDataAxis(size=10, scale=0.1, offset=10, units='mm')
>>> axis.scale_as_quantity
<Quantity(0.1, 'millimeter')>
>>> axis.convert_to_units('μm')
>>> axis.scale_as_quantity
<Quantity(100.0, 'micrometer')>
```

5.6 Axes storage and ordering

Note that HyperSpy rearranges the axes when compared to the array order. The following few paragraphs explain how and why.

Depending on how the array is arranged, some axes are faster to iterate than others. Consider an example of a book as the dataset in question. It is trivially simple to look at letters in a line, and then lines down the page, and finally pages in the whole book. However, if your words are written vertically, it can be inconvenient to read top-down (the lines are still horizontal, it's just the meaning that's vertical!). It is very time-consuming if every letter is on a different page, and for every word you have to turn 5-6 pages. Exactly the same idea applies here - in order to iterate through the data (most often for plotting, but for any other operation as well), you want to keep it ordered for “fast access”.

In Python (more explicitly *numpy*), the “fast axes order” is *C order* (also called row-major order). This means that the **last** axis of a *numpy* array is fastest to iterate over (i.e. the lines in the book). An alternative ordering convention is *F order* (column-major), where it is the other way round: the first axis of an array is the fastest to iterate over. In both cases, the further an axis is from the *fast axis* the slower it is to iterate over this axis. In the book analogy, you could think about reading the first lines of all pages, then the second and so on.

When data is acquired sequentially, it is usually stored in acquisition order. When a dataset is loaded, HyperSpy generally stores it in memory in the same order, which is good for the computer. However, HyperSpy will reorder and classify the axes to make it easier for humans. Let's imagine a single *numpy* array that contains pictures of a scene acquired with different exposure times on different days. In *numpy*, the array dimensions are (D, E, Y, X). This order makes it fast to iterate over the images in the order in which they were acquired. From a human point of view, this dataset is just a collection of images, so HyperSpy first classifies the image axes (X and Y) as *signal axes* and the remaining axes the *navigation axes*. Then it reverses the order of each set of axes because many humans are used to get the X axis first and, more generally, the axes in acquisition order from left to right. So, the same axes in HyperSpy are displayed like this: (E, D | X, Y).

Extending this to arbitrary dimensions, by default, we reverse the *numpy* axes, chop them into two chunks (signal and navigation), and then swap those chunks, at least when printing. As an example:

```
(a1, a2, a3, a4, a5, a6) # original (numpy)
(a6, a5, a4, a3, a2, a1) # reverse
(a6, a5) (a4, a3, a2, a1) # chop
(a4, a3, a2, a1) (a6, a5) # swap (HyperSpy)
```

In the background, HyperSpy also takes care of storing the data in memory in a “machine-friendly” way, so that iterating over the navigation axes is always fast.

5.7 Iterating over the AxesManager

One can iterate over the *AxesManager* to produce indices to the navigation axes. Each iteration will yield a new tuple of indices, sorted according to the iteration path specified in *iterpath*. Setting the *indices* property to a new index will update the accompanying signal so that signal methods that operate at a specific navigation index will now use that index, like `s.plot()`.

```
>>> s = hs.signals.Signal1D(np.zeros((2,3,10)))
>>> s.axes_manager.iterpath # check current iteration path
'serpentine'
>>> for index in s.axes_manager:
...     print(index)
(0, 0)
(1, 0)
```

(continues on next page)

(continued from previous page)

```
(2, 0)
(2, 1)
(1, 1)
(0, 1)
```

The *iterpath* attribute specifies the strategy that the *AxesManager* should use to iterate over the navigation axes. Two built-in strategies exist:

- 'serpentine' (default): starts at (0, 0), but when it reaches the final column (of index N), it continues from (1, N) along the next row, in the same way that a snake might slither, left and right.
- 'flyback': starts at (0, 0), continues down the row until the final column, “flies back” to the first column, and continues from (1, 0).

```
>>> s = hs.signals.Signal1D(np.zeros((2,3,10)))
>>> s.axes_manager.iterpath = 'flyback'
>>> for index in s.axes_manager:
...     print(index)
(0, 0)
(1, 0)
(2, 0)
(0, 1)
(1, 1)
(2, 1)
```

The *iterpath* can also be set using the *switch_iterpath()* context manager:

```
>>> s = hs.signals.Signal1D(np.zeros((2,3,10)))
>>> with s.axes_manager.switch_iterpath('flyback'):
...     for index in s.axes_manager:
...         print(index)
(0, 0)
(1, 0)
(2, 0)
(0, 1)
(1, 1)
(2, 1)
```

The *iterpath* can also be set to be a specific list of indices, like [(0,0), (0,1)], but can also be any generator of indices. Storing a high-dimensional set of indices as a list or array can take a significant amount of memory. By using a generator instead, one almost entirely removes such a memory footprint:

```
>>> s.axes_manager.iterpath = [(0,1), (1,1), (0,1)]
>>> for index in s.axes_manager:
...     print(index)
(0, 1)
(1, 1)
(0, 1)

>>> def reverse_flyback_generator():
...     for i in reversed(range(3)):
...         for j in reversed(range(2)):
...             yield (i,j)
```

(continues on next page)

(continued from previous page)

```
>>> s.axes_manager.iterpath = reverse_flyback_generator()
>>> for index in s.axes_manager:
...     print(index)
(2, 1)
(2, 0)
(1, 1)
(1, 0)
(0, 1)
(0, 0)
```

Since generators do not have a defined length, and does not need to include all navigation indices, a progressbar will be unable to determine how long it needs to be. To resolve this, a helper class can be imported that takes both a generator and a manually specified length as inputs:

```
>>> from hyperspy.axes import GeneratorLen
>>> gen = GeneratorLen(reverse_flyback_generator(), 6)
>>> s.axes_manager.iterpath = gen
```


SIGNAL1D TOOLS

The methods described in this section are only available for one-dimensional signals in the Signal1D class.

6.1 Cropping

The `crop_signal()` crops the the signal object along the signal axis (e.g. the spectral energy range) *in-place*. If no parameter is passed, a user interface appears in which to crop the one dimensional signal. For example:

```
>>> s = hs.data.two_gaussians()
>>> s.crop_signal(5, 15) # s is cropped in place
```

Additionally, cropping in HyperSpy can be performed using the *Signal indexing* syntax. For example, the following crops a signal to the 5.0-15.0 region:

```
>>> s = hs.data.two_gaussians()
>>> sc = s.isig[5.:15.] # s is not cropped, sc is a "cropped view" of s
```

It is possible to crop interactively using *Region Of Interest (ROI)*. For example:

```
>>> s = hs.data.two_gaussians()
>>> roi = hs.roi.SpanROI(left=5, right=15)
>>> s.plot()
>>> sc = roi.interactive(s)
```

6.2 Background removal

New in version 1.4: `zero_fill` and `plot_remainder` keyword arguments and big speed improvements.

The `remove_background()` method provides background removal capabilities through both a CLI and a GUI. The GUI displays an interactive preview of the remainder after background subtraction. Currently, the following background types are supported: Doniach, Exponential, Gaussian, Lorentzian, Polynomial, Power law (default), Offset, Skew normal, Split Voigt and Voigt. By default, the background parameters are estimated using analytical approximations (keyword argument `fast=True`). The fast option is not accurate for most background types - except Gaussian, Offset and Power law - but it is useful to estimate the initial fitting parameters before performing a full fit. For better accuracy, but higher processing time, the parameters can be estimated using curve fitting by setting `fast=False`.

Example of usage:

```
>>> s = exspy.data.EELS_MnFe(add_powerlaw=True)
>>> s.remove_background()
```

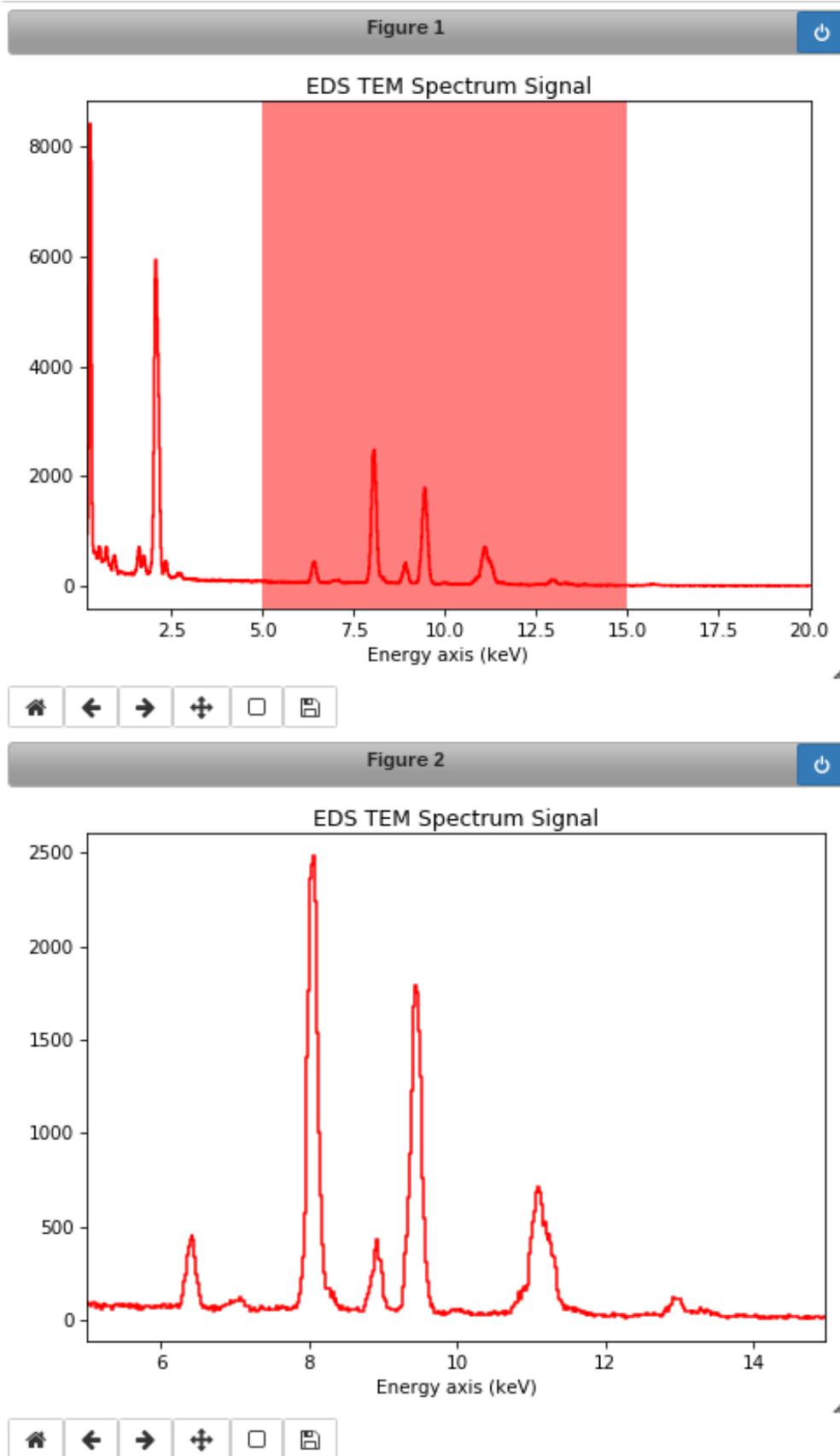


Fig. 1: Interactive spectrum cropping using a ROI.

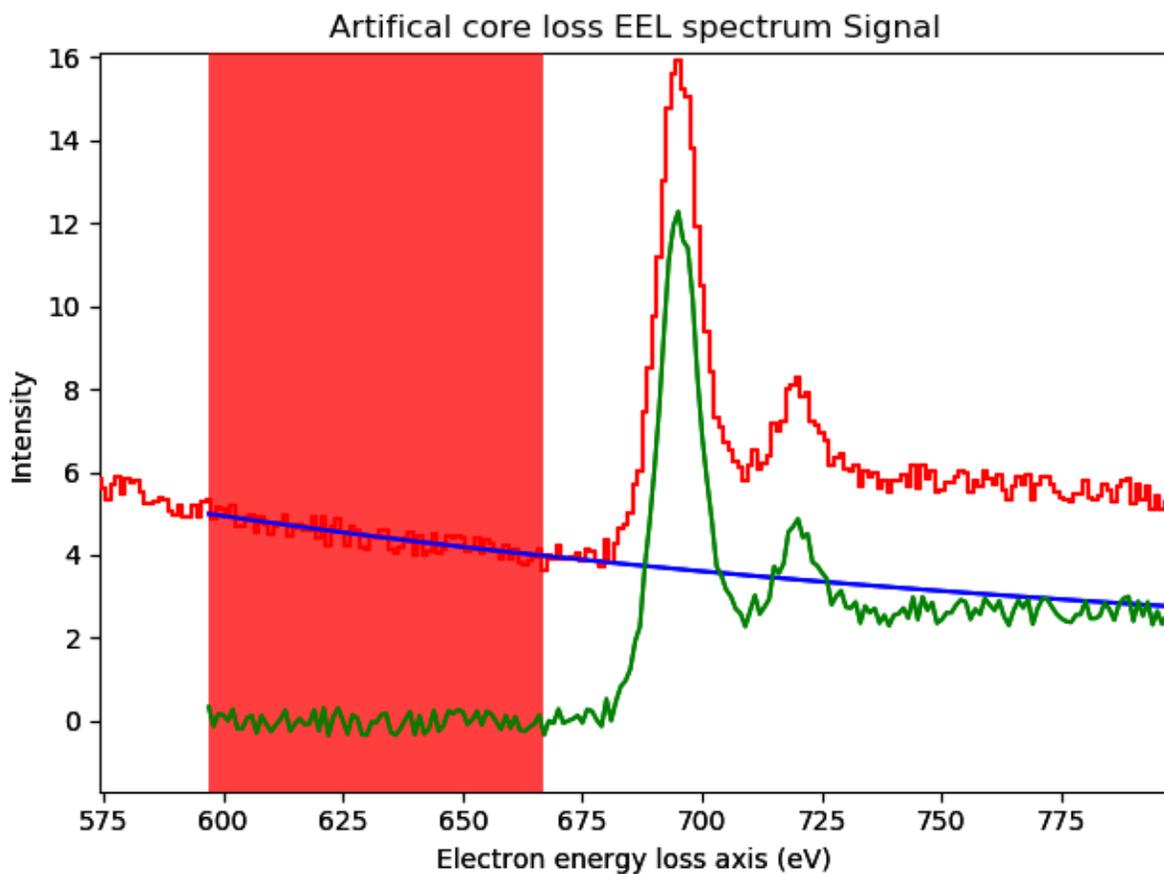


Fig. 2: Interactive background removal. In order to select the region used to estimate the background parameters (red area in the figure) click inside the axes of the figure and drag to the right without releasing the button.

6.3 Calibration

The `calibrate()` method provides a user interface to calibrate the spectral axis.

6.4 Alignment

The following methods use sub-pixel cross-correlation or user-provided shifts to align spectra. They support applying the same transformation to multiple files.

- `align1D()`
- `shift1D()`

6.5 Integration

To integrate signals use the `integrate1D()` method. Possibly in combination with a *Region Of Interest (ROI)* if interactivity is required. Otherwise, a signal subrange for integration can also be chosen with the `isig` method.

```
>>> s.isig[0.2:0.5].integrate1D(axis=0)
```

6.6 Data smoothing

The following methods (that include user interfaces when no arguments are passed) can perform data smoothing with different algorithms:

- `smooth_lowess()` (requires `statsmodels` to be installed)
- `smooth_tv()`
- `smooth_savitzky_golay()`

6.7 Spike removal

`spikes_removal_tool()` provides an user interface to remove spikes from spectra. The derivative histogram allows to identify the appropriate threshold. It is possible to use this tool on a specific interval of the data by *slicing the data*. For example, to use this tool in the signal between indices 8 and 17:

```
>>> s = hs.signals.Signal1D(np.arange(5*10*20).reshape((5, 10, 20)))
>>> s.isig[8:17].spikes_removal_tool()
```

The options `navigation_mask` or `signal_mask` provide more flexibility in the selection of the data, but these require a mask (boolean array) as parameter, which needs to be created manually:

```
>>> s = hs.signals.Signal1D(np.arange(5*10*20).reshape((5, 10, 20)))
```

To get a signal mask, get the mean over the navigation space

```
>>> s_mean = s.mean()
```

(continues on next page)

(continued from previous page)

```
>>> mask = s_mean > 495
>>> s.spikes_removal_tool(signal_mask=mask)
```

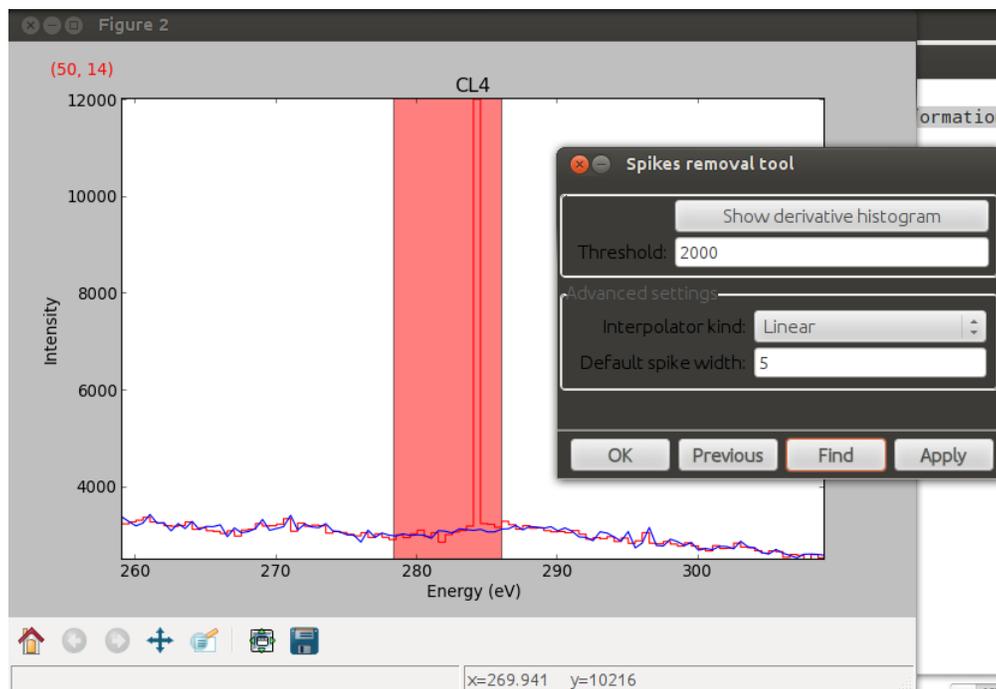


Fig. 3: Spikes removal tool.

6.8 Peak finding

A peak finding routine based on the work of T. O'Haver is available in HyperSpy through the `find_peaks1D_ohaver()` method.

6.9 Estimate peak width

For asymmetric peaks, *fitted functions* `<model.fitting>` may not provide an accurate description of the peak, in particular the peak width. The function `estimate_peak_width()` determines the width of a peak at a certain fraction of its maximum value.

6.10 Other methods

- Interpolate the spectra in between two positions *interpolate_in_between()*
- Convolve the spectra with a gaussian *gaussian_filter()*
- Apply a hanning taper to the spectra *hanning_taper()*

SIGNAL2D TOOLS

The methods described in this section are only available for two-dimensional signals in the *Signal2D* class.

7.1 Signal registration and alignment

The *align2D()* and *estimate_shift2D()* methods provide advanced image alignment functionality.

```
# Estimate shifts, then align the images
>>> shifts = s.estimate_shift2D() # doctest: +SKIP
>>> s.align2D(shifts=shifts) # doctest: +SKIP

# Estimate and align in a single step
>>> s.align2D() # doctest: +SKIP
```

Warning: *s.align2D()* will modify the data **in-place**. If you don't want to modify your original data, first take a copy before aligning.

Sub-pixel accuracy can be achieved in two ways:

- *scikit-image*'s upsampled matrix-multiplication DFT method [Guizar2008], by setting the *sub_pixel_factor* keyword argument
- for multi-dimensional datasets only, using the statistical method [Schaffer2004], by setting the reference keyword argument to "stat"

```
# skimage upsampling method
>>> shifts = s.estimate_shift2D(sub_pixel_factor=20) # doctest: +SKIP

# stat method
>>> shifts = s.estimate_shift2D(reference="stat") # doctest: +SKIP

# combined upsampling and statistical method
>>> shifts = s.estimate_shift2D(reference="stat", sub_pixel_factor=20) # doctest: +SKIP
```

If you have a large stack of images, the image alignment is automatically done in parallel.

You can control the number of threads used with the *num_workers* argument. Or by adjusting the scheduler of the *dask backend*.

```
# Estimate shifts
>>> shifts = s.estimate_shift2D() # doctest: +SKIP

# Align images in parallel using 4 threads
>>> s.align2D(shifts=shifts, num_workers=4) # doctest: +SKIP
```

7.2 Cropping a Signal2D

The `crop_signal()` method crops the image *in-place* e.g.:

```
>>> im = hs.data.wave_image()
>>> im.crop_signal(left=0.5, top=0.7, bottom=2.0) # im is cropped in-place
```

Cropping in HyperSpy is performed using the *Signal indexing* syntax. For example, to crop an image:

```
>>> im = hs.data.wave_image()
>>> # im is not cropped, imc is a "cropped view" of im
>>> imc = im.isig[0.5:, 0.7:2.0]
```

It is possible to crop interactively using *Region Of Interest (ROI)*. For example:

```
>>> im = hs.data.wave_image()
>>> roi = hs.roi.RectangularROI()
>>> im.plot()
>>> imc = roi.interactive(im)
>>> imc.plot()
```

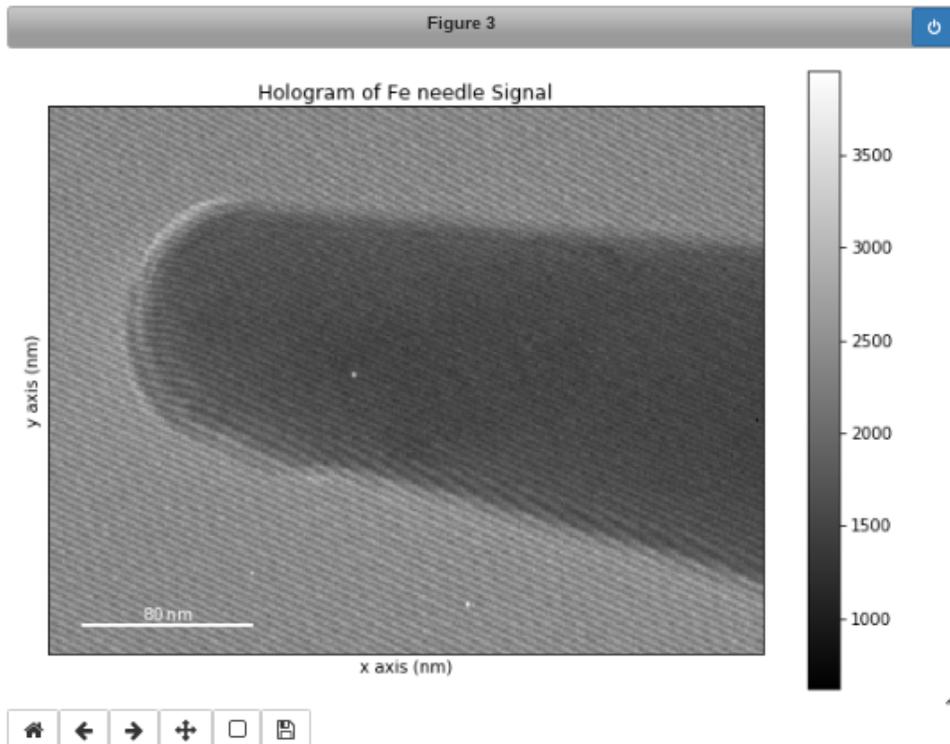
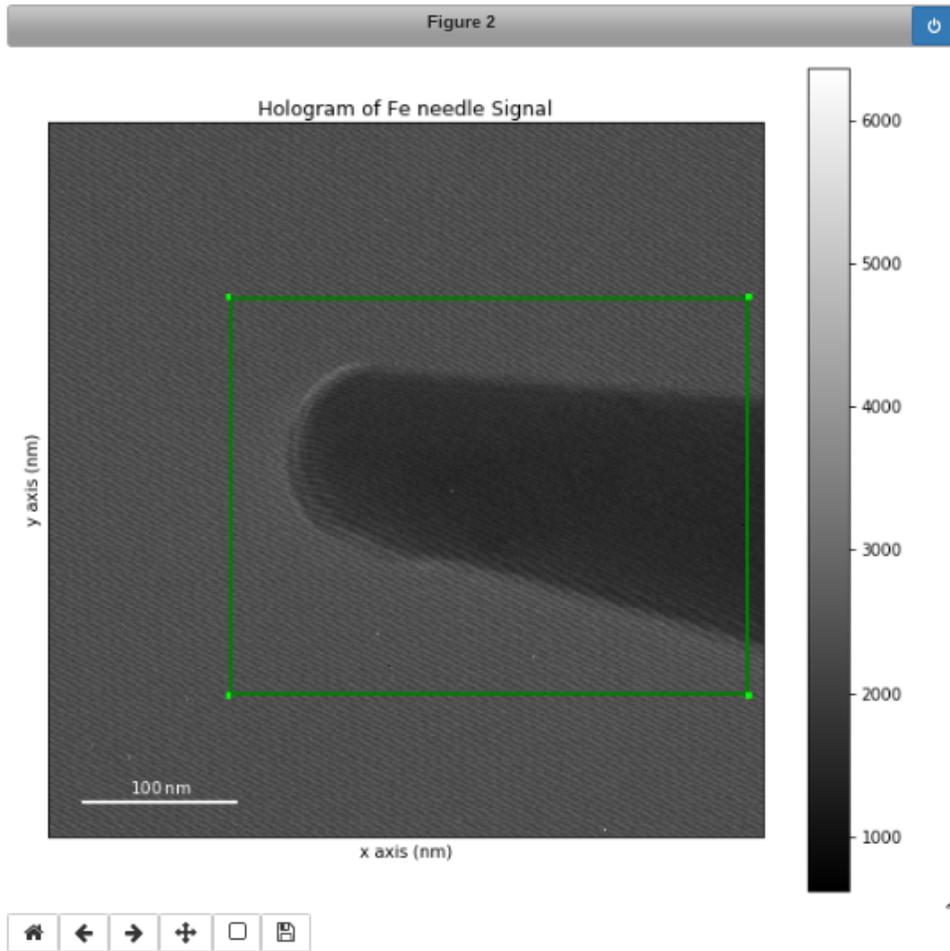
7.3 Interactive calibration

The scale can be calibrated interactively by using `calibrate()`, which is used to set the scale by dragging a line across some feature of known size.

```
>>> s = hs.signals.Signal2D(np.random.random((200, 200)))
>>> s.calibrate()
```

The same function can also be used non-interactively.

```
>>> s = hs.signals.Signal2D(np.random.random((200, 200)))
>>> s.calibrate(x0=1, y0=1, x1=5, y1=5, new_length=3.4, units="nm", interactive=False)
```



7.3. Interactive calibration Fig. 1: Interactive image cropping using a ROI.

7.4 Add a linear ramp

A linear ramp can be added to the signal via the `add_ramp()` method. The parameters `ramp_x` and `ramp_y` dictate the slope of the ramp in x- and y direction, while the offset is determined by the `offset` parameter. The fulcrum of the linear ramp is at the origin and the slopes are given in units of the axis with the according scale taken into account. Both are available via the `AxesManager` of the signal.

7.5 Peak finding

New in version 1.6.

The `find_peaks()` method provides access to a number of algorithms for peak finding in two dimensional signals. The methods available are:

7.5.1 Maximum based peak finder

```
>>> s.find_peaks(method='local_max')
>>> s.find_peaks(method='max')
>>> s.find_peaks(method='minmax')
```

These methods search for peaks using maximum (and minimum) values in the image. There all have a `distance` parameter to set the minimum distance between the peaks.

- the 'local_max' method uses the `skimage.feature.peak_local_max()` function (`distance` and `threshold` parameters are mapped to `min_distance` and `threshold_abs`, respectively).
- the 'max' method uses the `find_peaks_max()` function to search for peaks higher than $\alpha * \sigma$, where `alpha` is parameters and `sigma` is the standard deviation of the image. It also has a `distance` parameters to set the minimum distance between peaks.
- the 'minmax' method uses the `find_peaks_minmax()` function to locate the positive peaks in an image by comparing maximum and minimum filtered images. Its `threshold` parameter defines the minimum difference between the maximum and minimum filtered images.

7.5.2 Zaefferer peak finder

```
>>> s.find_peaks(method='zaefferer')
```

This algorithm was developed by Zaefferer [Zaefferer2000]. It is based on a gradient threshold followed by a local maximum search within a square window, which is moved until it is centered on the brightest point, which is taken as a peak if it is within a certain distance of the starting point. It uses the `find_peaks_zaefferer()` function, which can take `grad_threshold`, `window_size` and `distance_cutoff` as parameters. See the `find_peaks_zaefferer()` function documentation for more details.

7.5.3 Ball statistical peak finder

```
>>> s.find_peaks(method='stat')
```

Described by White [White2009], this method is based on finding points that have a statistically higher value than the surrounding areas, then iterating between smoothing and binarising until the number of peaks has converged. This method can be slower than the others, but is very robust to a variety of image types. It uses the `find_peaks_stat()` function, which can take `alpha`, `window_radius` and `convergence_ratio` as parameters. See the `find_peaks_stat()` function documentation for more details.

7.5.4 Matrix based peak finding

```
>>> s.find_peaks(method='laplacian_of_gaussians')
>>> s.find_peaks(method='difference_of_gaussians')
```

These methods are essentially wrappers around the Laplacian of Gaussian (`skimage.feature.blob_log()`) or the difference of Gaussian (`skimage.feature.blob_dog()`) methods, based on stacking the Laplacian/difference of images convolved with Gaussian kernels of various standard deviations. For more information, see the example in the [scikit-image documentation](#).

7.5.5 Template matching

```
>>> x, y = np.meshgrid(np.arange(-2, 2.5, 0.5), np.arange(-2, 2.5, 0.5))
>>> template = hs.model.components2D.Gaussian2D().function(x, y)
>>> s.find_peaks(method='template_matching', template=template, interactive=False)
<BaseSignal, title: , dimensions: (|ragged)>
```

This method locates peaks in the cross correlation between the image and a template using the `find_peaks_xc()` function. See the `find_peaks_xc()` function documentation for more details.

7.6 Interactive parametrization

Many of the peak finding algorithms implemented here have a number of tunable parameters that significantly affect their accuracy and speed. The GUIs can be used to set to select the method and set the parameters interactively:

```
>>> s.find_peaks(interactive=True)
```

Several widgets are available:

- The method selector is used to compare different methods. The last-set parameters are maintained.
- The parameter adjusters will update the parameters of the method and re-plot the new peaks.

Note: Some methods take significantly longer than others, particularly where there are a large number of peaks to be found. The plotting window may be inactive during this time.

▸ Navigation sliders

▼ Method parameters

Method	Local max	▼
Distance		3
Threshold		10.00

Compute over navi... Close

DATA VISUALIZATION

The object returned by `load()`, a `BaseSignal` instance, has a `plot()` method that is powerful and flexible to visualize n-dimensional data. In this chapter, the visualisation of multidimensional data is exemplified with two experimental datasets: an EELS spectrum image and an EDX dataset consisting of a secondary electron emission image stack and a 3D hyperspectral image, both simultaneously acquired by recording two signals in parallel in a FIB/SEM.

```
>>> s = hs.load('YourDataFilenameHere')
>>> s.plot()
```

if the object is single spectrum or an image one window will appear when calling the plot method.

8.1 Multidimensional spectral data

If the object is a 1D or 2D spectrum-image (i.e. with 2 or 3 dimensions when including energy) two figures will appear, one containing a plot of the spectrum at the current coordinates and the other an image of the data summed over its spectral dimension if 2D or an image with the spectral dimension in the x-axis if 1D:

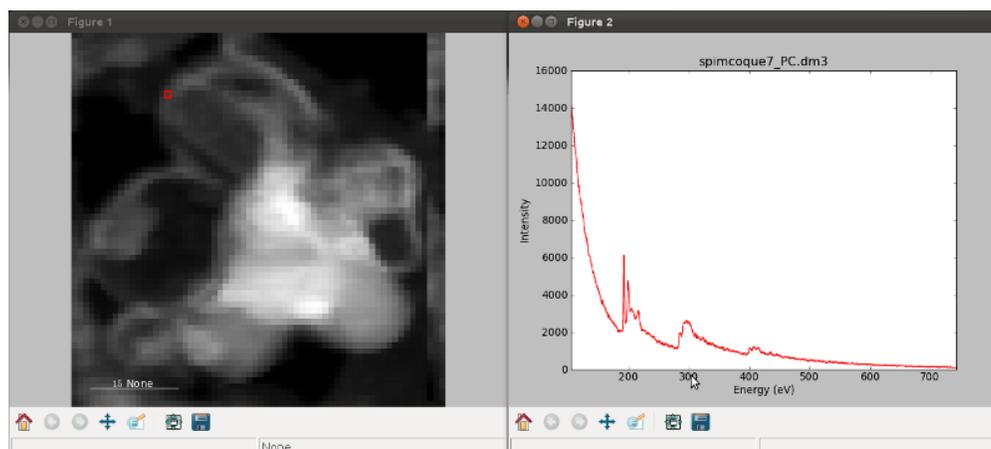


Fig. 1: Visualisation of a 2D spectrum image.

New in version 1.4: Customizable keyboard shortcuts to navigate multi-dimensional datasets.

To change the current coordinates, click on the pointer (which will be a line or a square depending on the dimensions of the data) and drag it around. It is also possible to move the pointer by using the numpad arrows **when numlock is on and the spectrum or navigator figure is selected**. When using the numpad arrows the PageUp and PageDown keys change the size of the step.

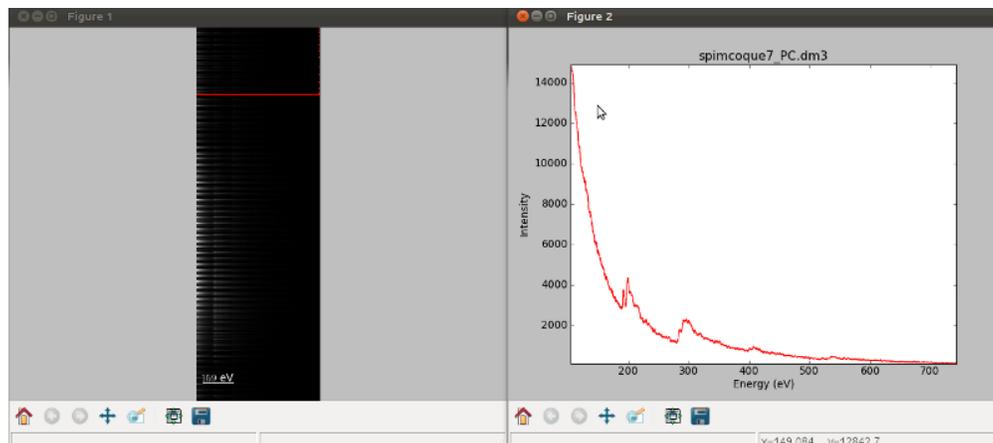


Fig. 2: Visualisation of a 1D spectrum image.

The current coordinates can be either set by navigating the `plot()`, or specified by pixel indices in `s.axes_manager.indices` or as calibrated coordinates in `s.axes_manager.coordinates`.

An extra cursor can be added by pressing the `e` key. Pressing `e` once more will disable the extra cursor:

In `matplotlib`, left and right arrow keys are by default set to navigate the “zoom” history. To avoid the problem of changing zoom while navigating, `Ctrl + arrows` can be used instead. Navigating without using the modifier keys will be deprecated in version 2.0.

To navigate navigation dimensions larger than 2, modifier keys can be used. The defaults are `Shift + left/right` and `Shift + up/down`, (`Alt + left/right` and `Alt + up/down`) for navigating dimensions 2 and 3 (4 and 5) respectively. Modifier keys do not work with the numpad.

Hotkeys and modifier keys for navigating the plot can be set in the *HyperSpy plot preferences*. Note that some combinations will not work for all platforms, as some systems reserve them for other purposes.

If you want to jump to some point in the dataset. In that case you can hold the `Shift` key and click the point you are interested in. That will automatically take you to that point in the data. This also helps with lazy data as you don’t have to load every chunk in between.

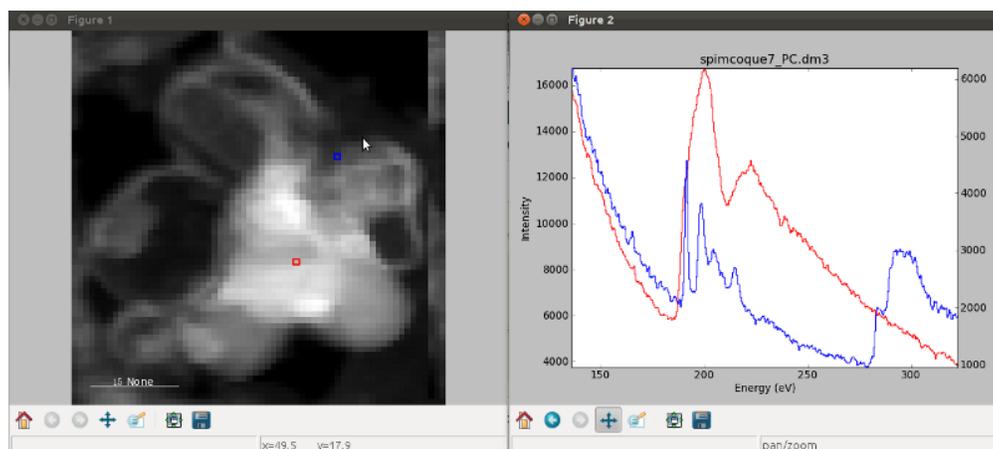


Fig. 3: Visualisation of a 2D spectrum image using two pointers.

Sometimes the default size of the rectangular cursors used to navigate images can be too small to be dragged or even seen. It is possible to change the size of the cursors by pressing the `+` and `-` keys **when the navigator window is**

selected.

The following keyboard shortcuts are available when the 1D signal figure is in focus:

Table 1: Keyboard shortcuts available on the signal figure of 1D signal data

key	function
e	Switch second pointer on/off
Ctrl + Arrows	Change coordinates for dimensions 0 and 1 (typically x and y)
Shift + Arrows	Change coordinates for dimensions 2 and 3
Alt + Arrows	Change coordinates for dimensions 4 and 5
PageUp	Increase step size
PageDown	Decrease step size
+	Increase pointer size when the navigator is an image
-	Decrease pointer size when the navigator is an image
l	switch the scale of the y-axis between logarithmic and linear

To close all the figures run the following command:

```
>>> import matplotlib.pyplot as plt
>>> plt.close('all')
```

Note: `plt.close('all')` is a `matplotlib` command. `Matplotlib` is the library that `HyperSpy` uses to produce the plots. You can learn how to pan/zoom and more [in the matplotlib documentation](#)

Note: Plotting `float16` images is currently not supported by `matplotlib`; however, it is possible to convert the type of the data by using the `change_dtype()` method, e.g. `s.change_dtype('float32')`.

8.2 Multidimensional image data

Equivalently, if the object is a 1D or 2D image stack two figures will appear, one containing a plot of the image at the current coordinates and the other a spectrum or an image obtained by summing over the image dimensions:

New in version 1.4: 1 keyboard shortcut

The following keyboard shortcuts are available when the 2D signal figure is in focus:

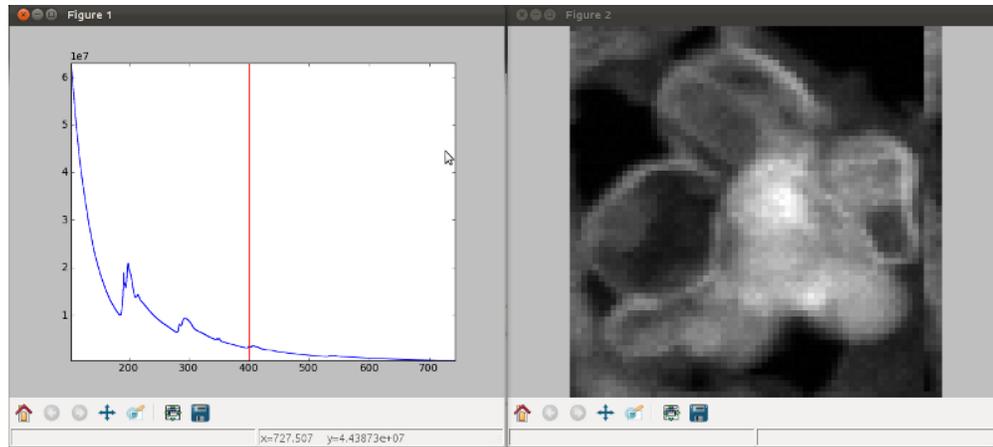


Fig. 4: Visualisation of a 1D image stack.

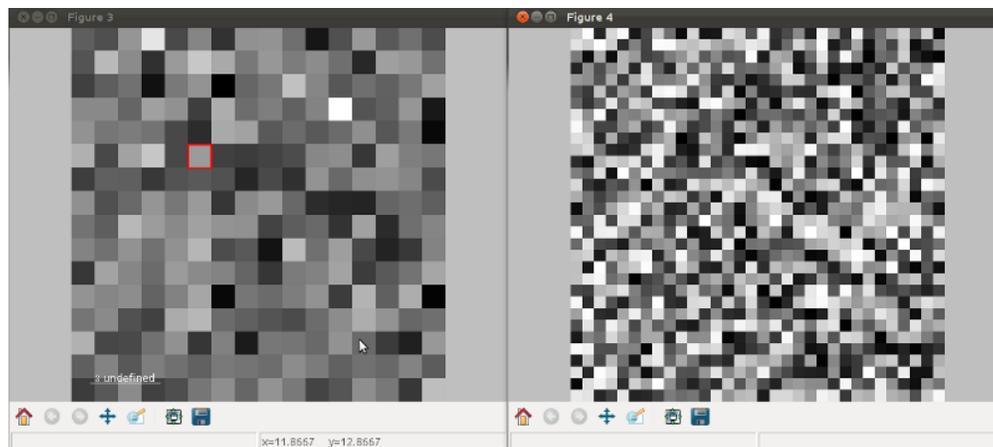


Fig. 5: Visualisation of a 2D image stack.

Table 2: Keyboard shortcuts available on the signal figure of 2D signal data

key	function
Ctrl + Arrows	Change coordinates for dimensions 0 and 1 (typically x and y)
Shift + Arrows	Change coordinates for dimensions 2 and 3
Alt + Arrows	Change coordinates for dimensions 4 and 5
PageUp	Increase step size
PageDown	Decrease step size
+	Increase pointer size when the navigator is an image
-	Decrease pointer size when the navigator is an image
h	Launch the contrast adjustment tool
l	switch the norm of the intensity between logarithmic and linear

8.3 Customising image plot

The image plot can be customised by passing additional arguments when plotting. Colorbar, scalebar and contrast controls are HyperSpy-specific, however `matplotlib.axes.Axes.imshow()` arguments are supported as well:

```
>>> import scipy
>>> img = hs.signals.Signal2D(scipy.datasets.ascent())
>>> img.plot(colorbar=True, scalebar=False, axes_ticks=True, cmap='RdYlBu_r')
```

New in version 1.4: `norm` keyword argument

The `norm` keyword argument can be used to select between linear, logarithmic or custom (using a matplotlib norm) intensity scale. The default, “auto”, automatically selects a logarithmic scale when plotting a power spectrum.

New in version 1.6: `autoscale` keyword argument

The `autoscale` keyword argument can be used to specify which axis limits are reset when the data or navigation indices change. It can take any combinations of the following characters:

- 'x': to reset the horizontal axes
- 'y': to reset the vertical axes
- 'v': to reset the contrast of the image according to `vmin` and `vmax`

By default (`autoscale='v'`), only the contrast of the image will be reset automatically. For example, to reset the extent of the image (x and y) to their maxima but not the contrast, use `autoscale='xy'`; To reset all limits, including the contrast of the image, use `autoscale='xyv'`:

```
>>> import numpy as np
>>> img = hs.signals.Signal2D(np.arange(10*10*10).reshape(10, 10, 10))
>>> img.plot(autoscale='xyv')
```

When plotting using divergent colormaps, if `centre_colormap` is `True` (default) the contrast is automatically adjusted so that zero corresponds to the center of the colormap (usually white). This can be useful e.g. when displaying images that contain pixels with both positive and negative values.

The following example shows the effect of centring the color map:

```
>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 128)
>>> xx, yy = np.meshgrid(x, x)
```

(continues on next page)

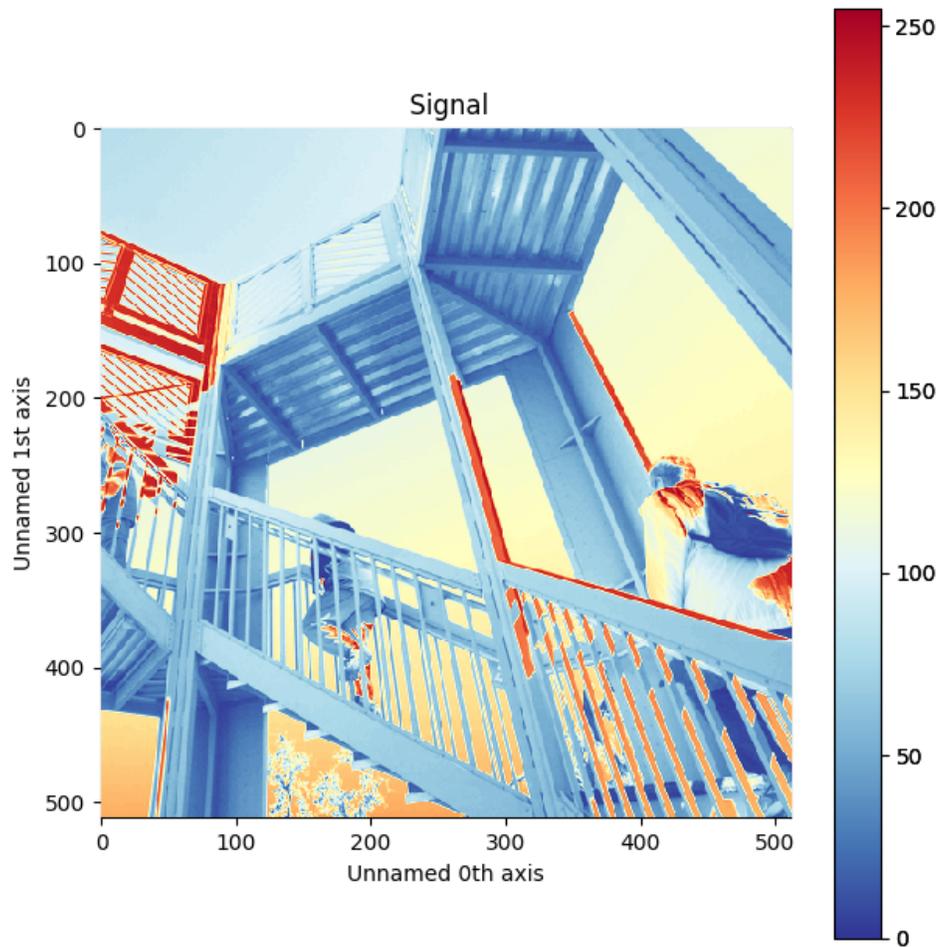


Fig. 6: Custom colormap and switched off scalebar in an image.

(continued from previous page)

```

>>> data1 = np.sin(xx * yy)
>>> data2 = data1.copy()
>>> data2[data2 < 0] /= 4
>>> im = hs.signals.Signal2D([data1, data2])
>>> hs.plot.plot_images(im, cmap="RdBu", tight_layout=True)
[<Axes: title={'center': ' (0,)'}>, <Axes: title={'center': ' (1,)'>]

```

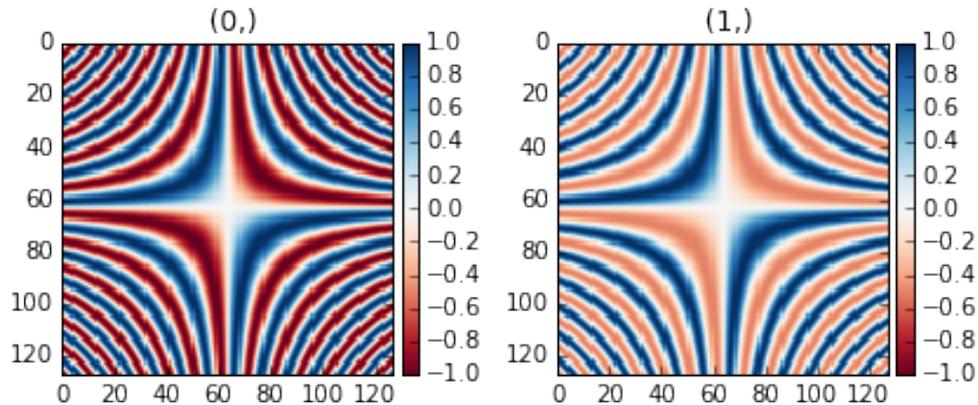


Fig. 7: Divergent color map with Centre colormap enabled (default).

The same example with the feature disabled:

```

>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 128)
>>> xx, yy = np.meshgrid(x, x)
>>> data1 = np.sin(xx * yy)
>>> data2 = data1.copy()
>>> data2[data2 < 0] /= 4
>>> im = hs.signals.Signal2D([data1, data2])
>>> hs.plot.plot_images(im, centre_colormap=False, cmap="RdBu", tight_layout=True)
[<Axes: title={'center': ' (0,)'>, <Axes: title={'center': ' (1,)'>]

```

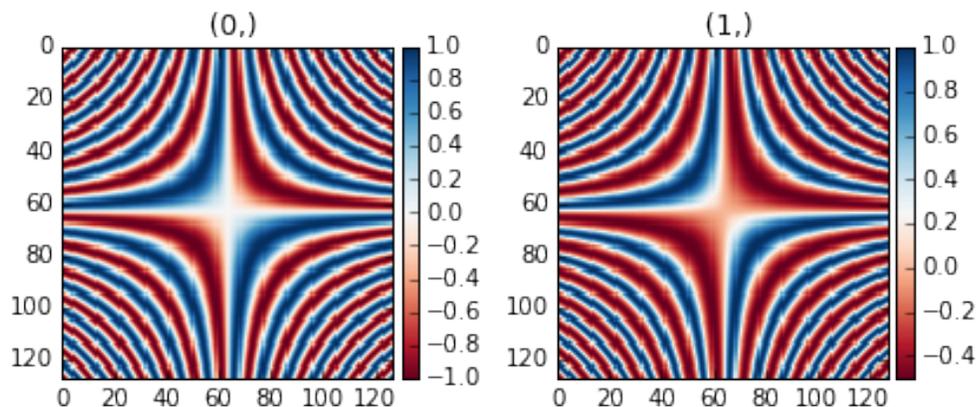


Fig. 8: Divergent color map with centre_colormap disabled.

New in version 2.0.0: `plot_style` keyword argument to allow for “horizontal” or “vertical” alignment of subplots (e.g.

navigator and signal) when using the *ipyml* or *widget* backends. A default value can also be set using the *HyperSpy plot preferences*.

8.4 Customizing the “navigator”

New in version 1.1.2: Passing keyword arguments to the navigator plot.

The navigator can be customised by using the `navigator_kwds` argument. For example, in case of an image navigator, all image plot arguments mentioned in the section *Customising image plot* can be passed as a dictionary to the `navigator_kwds` argument:

```
>>> import numpy as np
>>> import scipy
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> ims = hs.signals.BaseSignal(np.random.rand(15,13)).T * im
>>> ims.metadata.General.title = 'My Images'
>>> ims.plot(colorbar=False,
...         scalebar=False,
...         axes_ticks=False,
...         cmap='viridis',
...         navigator_kwds=dict(colorbar=True,
...                             scalebar_color='red',
...                             cmap='Blues',
...                             axes_ticks=False)
...     )
```

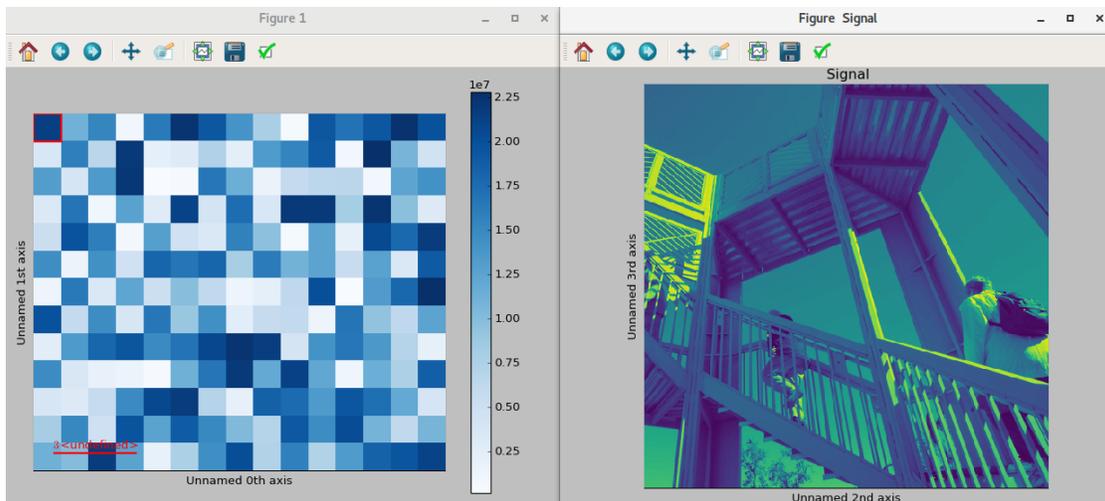


Fig. 9: Custom different options for both signal and navigator image plots

Data files used in the following examples can be downloaded using

```
>>> #Download the data (130MB)
>>> from urllib.request import urlretrieve, urlopen
>>> from zipfile import ZipFile
>>> files = urlretrieve("https://www.dropbox.com/s/s7cx92mfh2zvt3x/"
...                   "HyperSpy_demos_EDX_SEM_files.zip?raw=1",
```

(continues on next page)

(continued from previous page)

```

...         "./HyperSpy_demos_EDX_SEM_files.zip")
>>> with ZipFile("HyperSpy_demos_EDX_SEM_files.zip") as z:
...     z.extractall()

```

Note: See also the [SEM EDS tutorials](#) .

Note: The sample and the data used in this chapter are described in [\[Burdet2013\]](#).

Stack of 2D images can be imported as an 3D image and plotted with a slider instead of the 2D navigator as in the previous example.

```

>>> img = hs.load('Ni_superalloy_0*.tif', stack=True)
>>> img.plot(navigator='slider')

```

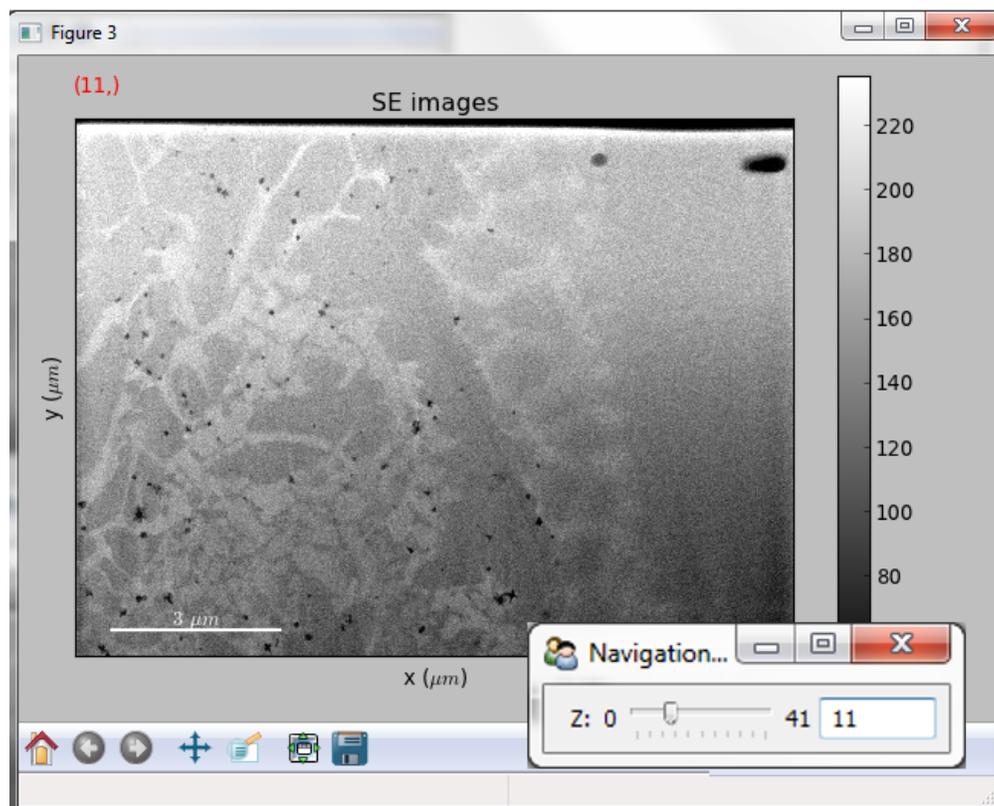


Fig. 10: Visualisation of a 3D image with a slider.

A stack of 2D spectrum images can be imported as a 3D spectrum image and plotted with sliders.

```

>>> s = hs.load('Ni_superalloy_0*.rpl', stack=True).as_signal1D(0)
>>> s.plot()

```

If the 3D images has the same spatial dimension as the 3D spectrum image, it can be used as an external signal for the navigator.

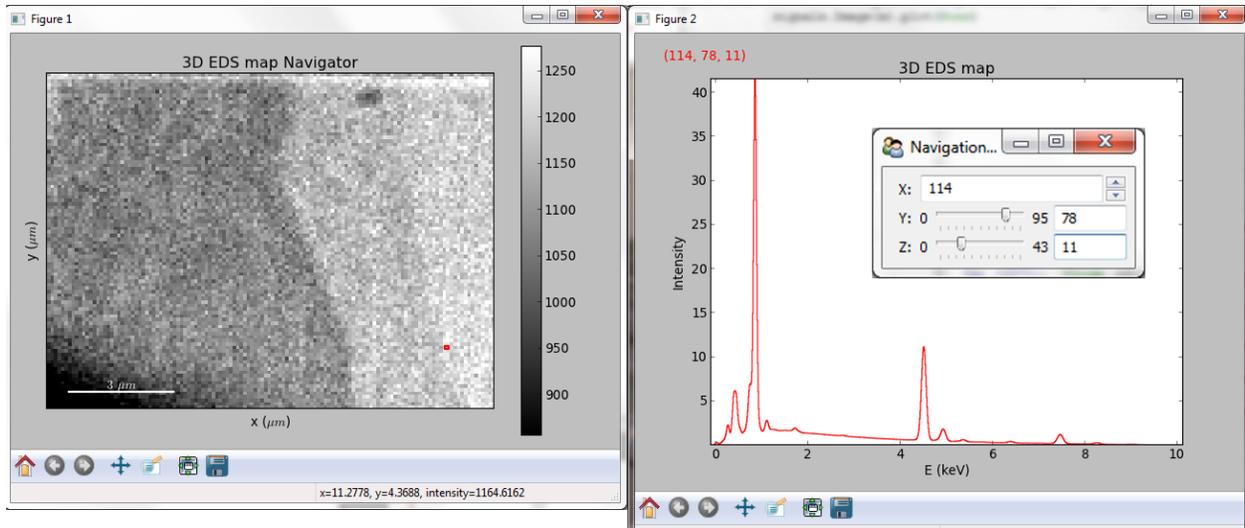


Fig. 11: Visualisation of a 3D spectrum image with sliders.

```
>>> im = hs.load('Ni_superalloy_0*.tif', stack=True)
>>> s = hs.load('Ni_superalloy_0*.rpl', stack=True).as_signal1D(0)
>>> dim = s.axes_manager.navigation_shape
```

Rebin the image

```
>>> im = im.rebin([dim[2], dim[0], dim[1]])
>>> s.plot(navigator=im)
```

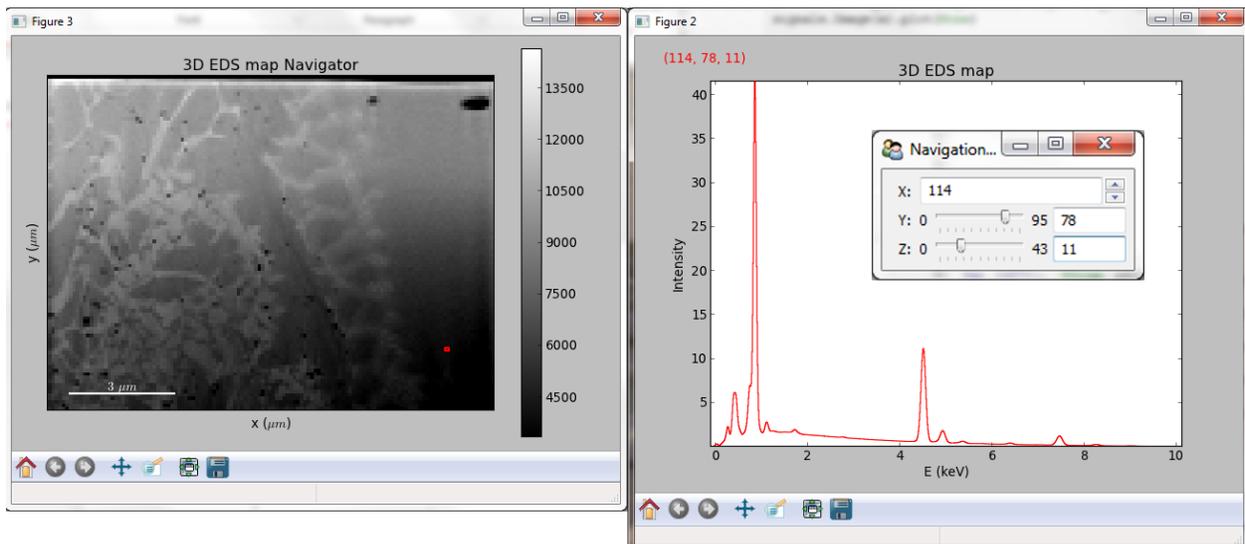


Fig. 12: Visualisation of a 3D spectrum image. The navigator is an external signal.

The 3D spectrum image can be transformed in a stack of spectral images for an alternative display.

```
>>> imgSpec = hs.load('Ni_superalloy_0*.rpl', stack=True)
```

(continues on next page)

(continued from previous page)

```
>>> imgSpec.plot(navigator='spectrum')
```

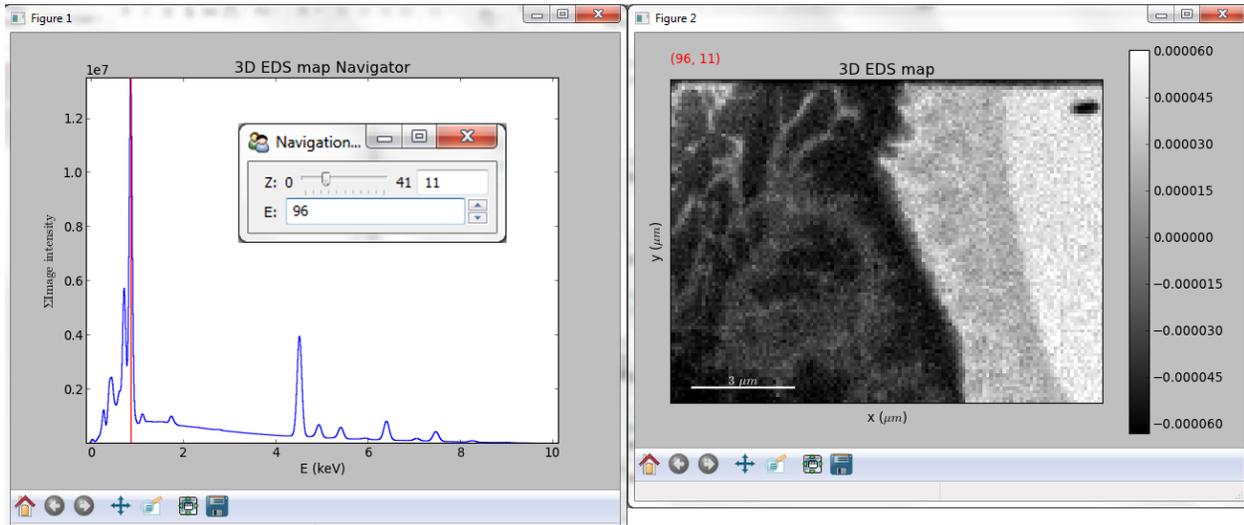


Fig. 13: Visualisation of a stack of 2D spectral images.

An external signal (e.g. a spectrum) can be used as a navigator, for example the “maximum spectrum” for which each channel is the maximum of all pixels.

```
>>> imgSpec = hs.load('Ni_superalloy_0*.rpl', stack=True)
>>> specMax = imgSpec.max(-1).max(-1).max(-1).as_signal1D(0)
>>> imgSpec.plot(navigator=specMax)
```

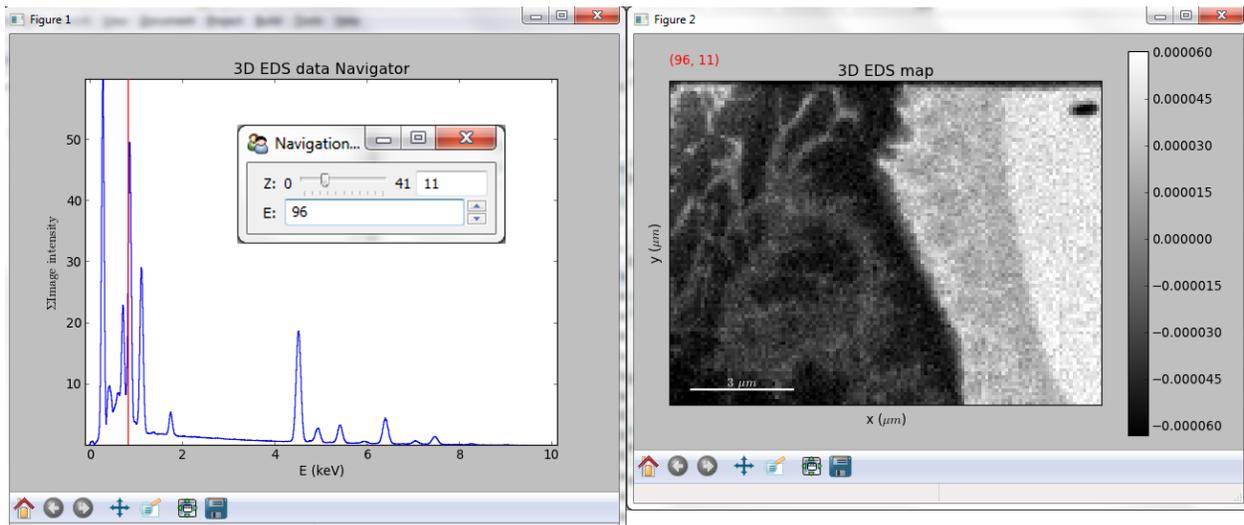


Fig. 14: Visualisation of a stack of 2D spectral images. The navigator is the “maximum spectrum”.

Lastly, if no navigator is needed, “navigator=None” can be used.

8.5 Using Mayavi to visualize 3D data

Data files used in the following examples can be downloaded using

```
>>> from urllib.request import urlretrieve
>>> url = 'http://cook.msm.cam.ac.uk/~hyperspy/EDS_tutorial/'
>>> urlretrieve(url + 'Ni_La_intensity.hdf5', 'Ni_La_intensity.hdf5')
```

Note: See also the [EDS tutorials](#) .

Although HyperSpy does not currently support plotting when `signal_dimension` is greater than 2, *Mayavi* can be used for this purpose.

In the following example we also use *scikit-image* for noise reduction. More details about `exspy.signals.EDSSpectrum.get_lines_intensity()` method can be found in [EDS lines intensity](#).

```
>>> from mayavi import mlab
>>> ni = hs.load('Ni_La_intensity.hdf5')
>>> mlab.figure()
>>> mlab.contour3d(ni.data, contours=[85])
>>> mlab.outline(color=(0, 0, 0))
```

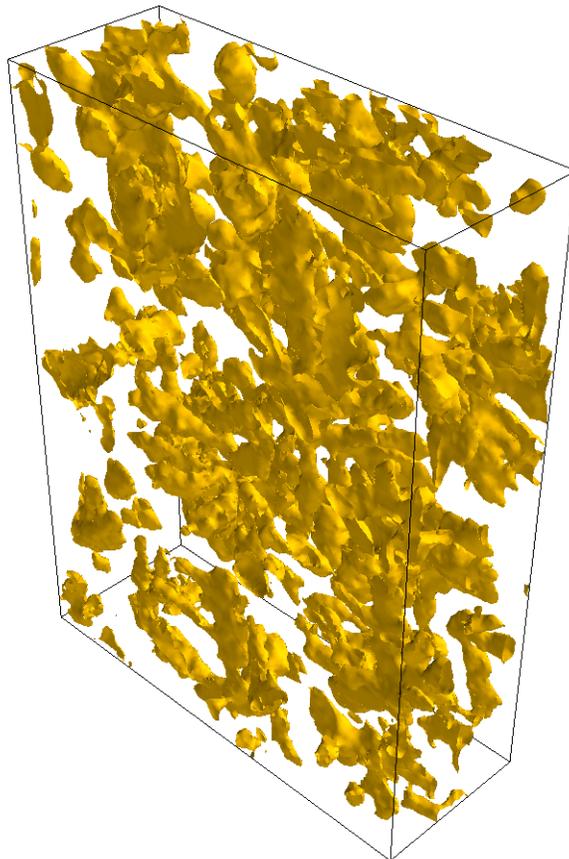


Fig. 15: Visualisation of isosurfaces with mayavi.

Note: See also the [SEM EDS tutorials](#) .

Note: The sample and the data used in this chapter are described in P. Burdet, *et al.*, Ultramicroscopy, 148, p. 158-167 (2015).

8.6 Plotting multiple signals

HyperSpy provides three functions to plot multiple signals (spectra, images or other signals): `plot_images()`, `plot_spectra()`, and `plot_signals()` in the `plot` package.

8.6.1 Plotting several images

`plot_images()` is used to plot several images in the same figure. It supports many configurations and has many options available to customize the resulting output. The function returns a list of `matplotlib.axes.Axes`, which can be used to further customize the figure. Some examples are given below. Plots generated from another installation may look slightly different due to matplotlib GUI backends and default font sizes. To change the font size globally, use the command `matplotlib.rcParams.update({'font.size': 8})`.

New in version 1.5: Add support for plotting `BaseSignal` with navigation dimension 2 and signal dimension 0.

A common usage for `plot_images()` is to view the different slices of a multidimensional image (a *hyperimage*):

```
>>> import scipy
>>> image = hs.signals.Signal2D([scipy.datasets.ascent()]*6)
>>> angles = hs.signals.BaseSignal(range(10,70,10))
>>> image.map(scipy.ndimage.rotate, angle=angles.T, reshape=False)
>>> hs.plot.plot_images(image, tight_layout=True)
```

This example is explained in *Signal iterator*.

By default, `plot_images()` will attempt to auto-label the images based on the Signal titles. The labels (and title) can be customized with the `suptitle` and `label` arguments. In this example, the axes labels and the ticks are also disabled with `axes_decor`:

```
>>> import scipy
>>> image = hs.signals.Signal2D([scipy.datasets.ascent()]*6)
>>> angles = hs.signals.BaseSignal(range(10,70,10))
>>> image.map(scipy.ndimage.rotate, angle=angles.T, reshape=False)
>>> hs.plot.plot_images(
...     image, suptitle='Turning Ascent', axes_decor='off',
...     label=['Rotation {}$^\degree$'.format(angles.data[i]) for
...           i in range(angles.data.shape[0])], colorbar=None)
```

`plot_images()` can also be used to easily plot a list of `Images`, comparing different `Signals`, including RGB images. This example also demonstrates how to wrap labels using `labelwrap` (for preventing overlap) and using a single `colorbar` for all the `Images`, as opposed to multiple individual ones:

```
>>> import scipy
>>> import numpy as np
```

(continues on next page)

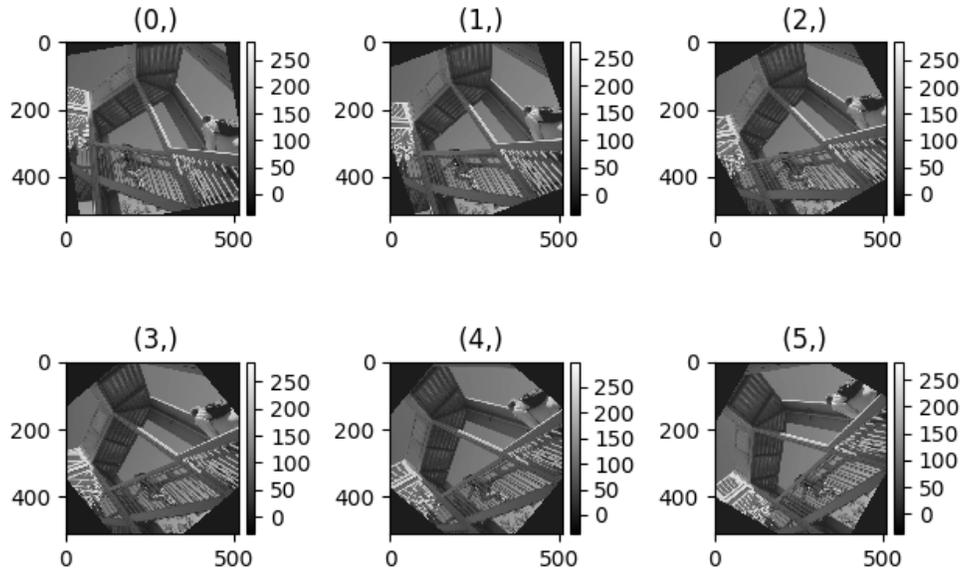


Fig. 16: Figure generated with `plot_images()` using the default values.

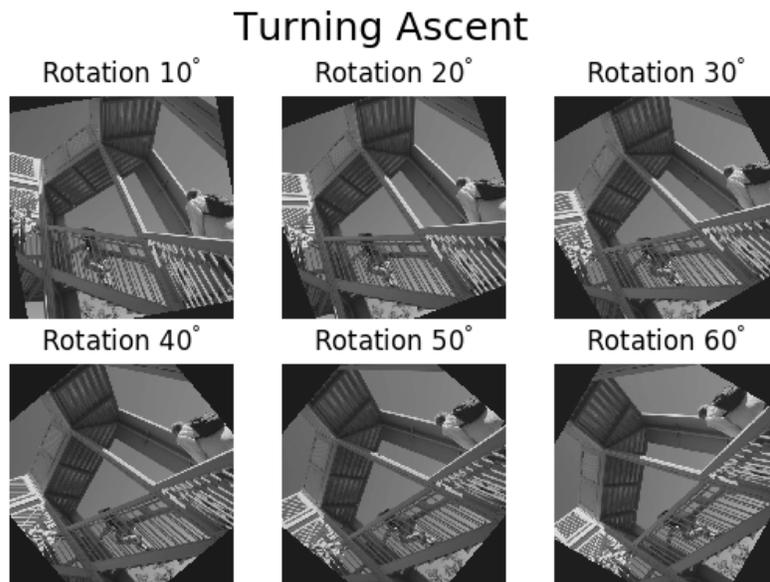


Fig. 17: Figure generated with `plot_images()` with customised labels.

(continued from previous page)

```

>>>

Load red channel of raccoon as an image

>>> image0 = hs.signals.Signal2D(scipy.datasets.face()[ :, :, 0])
>>> image0.metadata.General.title = 'Rocky Raccoon - R'

Load ascent into a length 6 hyper-image

>>> image1 = hs.signals.Signal2D([scipy.datasets.ascent()*6)
>>> angles = hs.signals.BaseSignal(np.arange(10,70,10)).T
>>> image1.map(scipy.ndimage.rotate, angle=angles, reshape=False)
>>> image1.data = np.clip(image1.data, 0, 255) # clip data to int range

Load green channel of raccoon as an image

>>> image2 = hs.signals.Signal2D(scipy.datasets.face()[ :, :, 1])
>>> image2.metadata.General.title = 'Rocky Raccoon - G'
>>>

Load rgb image of the raccoon

>>> rgb = hs.signals.Signal1D(scipy.datasets.face())
>>> rgb.change_dtype("rgb8")
>>> rgb.metadata.General.title = 'Raccoon - RGB'
>>>

>>> images = [image0, image1, image2, rgb]
>>> for im in images:
...     ax = im.axes_manager.signal_axes
...     ax[0].name, ax[1].name = 'x', 'y'
...     ax[0].units, ax[1].units = 'mm', 'mm'
>>> hs.plot.plot_images(images, tight_layout=True,
...                     colorbar='single', labelwrap=20)

```

Data files used in the following example can be downloaded using (These data are described in [Rossouw2015]).

```

>>> #Download the data (1MB)
>>> from urllib.request import urlretrieve, urllopen
>>> from zipfile import ZipFile
>>> files = urlretrieve("https://www.dropbox.com/s/ecdlgwxjq04m5mx/"
...                    "HyperSpy_demos_EDS_TEM_files.zip?raw=1",
...                    "./HyperSpy_demos_EDX_TEM_files.zip")
>>> with ZipFile("HyperSpy_demos_EDX_TEM_files.zip") as z:
...     z.extractall()

```

Another example for this function is plotting EDS line intensities see [EDS chapter](#). One can use the following commands to get a representative figure of the X-ray line intensities of an EDS spectrum image. This example also demonstrates changing the colormap (with *cmap*), adding scalebars to the plots (with *scalebar*), and changing the *padding* between the images. The padding is specified as a dictionary, which is passed to `matplotlib.figure.Figure.subplots_adjust()`.

```

>>> si_EDS = hs.load("core_shell.hdf5")
>>> im = si_EDS.get_lines_intensity()

```

(continues on next page)

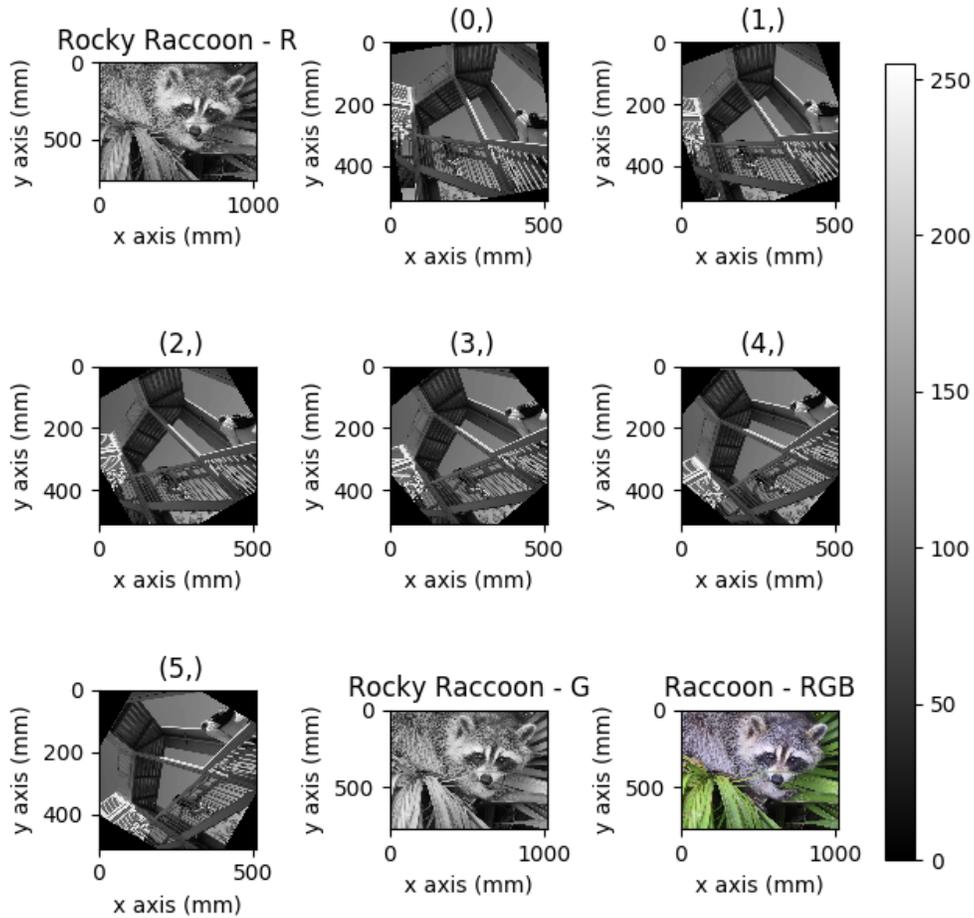


Fig. 18: Figure generated with `plot_images()` from a list of images.

(continued from previous page)

```
>>> hs.plot.plot_images(im,
...     tight_layout=True, cmap='RdYlBu_r', axes_decor='off',
...     colorbar='single', vmin='1th', vmax='99th', scalebar='all',
...     scalebar_color='black', suptitle_fontsize=16,
...     padding={'top':0.8, 'bottom':0.10, 'left':0.05,
...             'right':0.85, 'wspace':0.20, 'hspace':0.10})
```

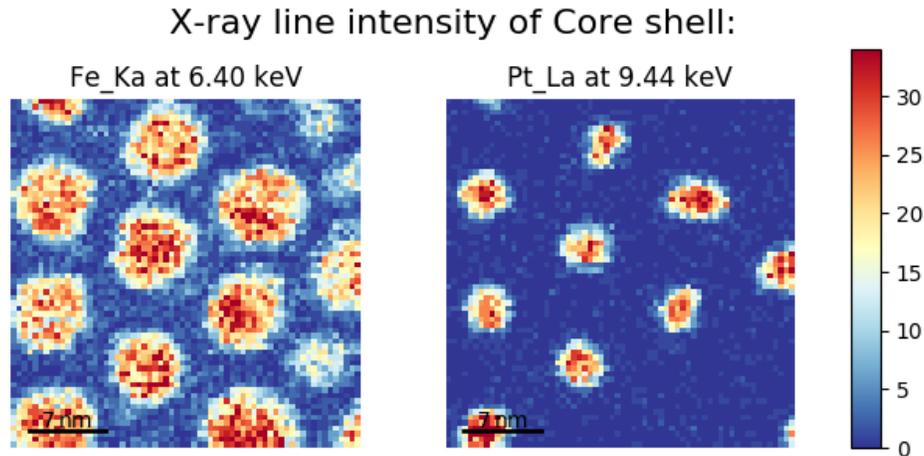


Fig. 19: Using `plot_images()` to plot the output of `get_lines_intensity()`.

Note: This padding can also be changed interactively by clicking on the  button in the GUI (button may be different when using different graphical backends).

Finally, the `cmap` option of `plot_images()` supports iterable types, allowing the user to specify different colormaps for the different images that are plotted by providing a list or other generator:

```
>>> si_EDS = hs.load("core_shell.hdf5")
>>> im = si_EDS.get_lines_intensity()
>>> hs.plot.plot_images(im,
...     tight_layout=True, cmap=['viridis', 'plasma'], axes_decor='off',
...     colorbar='multi', vmin='1th', vmax='99th', scalebar=[0],
...     scalebar_color='white', suptitle_fontsize=16)
```

The `cmap` argument can also be given as `'mpl_colors'`, and as a result, the images will be plotted with colormaps generated from the default matplotlib colors, which is very helpful when plotting multiple spectral signals and their relative intensities (such as the results of a `decomposition()` analysis). This example uses `plot_spectra()`, which is explained in the [next section](#).

```
>>> si_EDS = hs.load("core_shell.hdf5")
>>> si_EDS.change_dtype('float')
>>> si_EDS.decomposition(True, algorithm='NMF', output_dimension=3)
>>> factors = si_EDS.get_decomposition_factors()
>>>
>>> # the first factor is a very strong carbon background component, so we
```

(continues on next page)

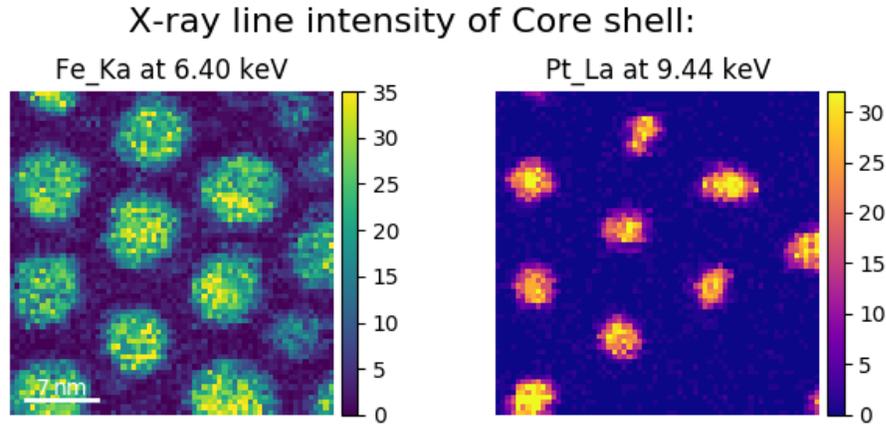


Fig. 20: Using `plot_images()` to plot the output of `get_lines_intensity()` using a unique colormap for each image.

(continued from previous page)

```
>>> # normalize factor intensities for easier qualitative comparison
>>> for f in factors:
...     f.data /= f.data.max()
>>>
>>> loadings = si_EDS.get_decomposition_loadings()
>>>
>>> hs.plot.plot_spectra(factors.isig[:14.0], style='cascade', padding=-1)
>>>
>>> # add some lines to nicely label the peak positions
>>> plt.axvline(6.403, c='C2', ls=':', lw=0.5)
>>> plt.text(x=6.503, y=0.85, s='Fe-K$\alpha$', color='C2')
>>> plt.axvline(9.441, c='C1', ls=':', lw=0.5)
>>> plt.text(x=9.541, y=0.85, s='Pt-L$\alpha$', color='C1')
>>> plt.axvline(2.046, c='C1', ls=':', lw=0.5)
>>> plt.text(x=2.146, y=0.85, s='Pt-M', color='C1')
>>> plt.axvline(8.040, ymax=0.8, c='k', ls=':', lw=0.5)
>>> plt.text(x=8.14, y=0.35, s='Cu-K$\alpha$', color='k')
>>>
>>> hs.plot.plot_images(loadings, cmap='mpl_colors',
...                     axes_decor='off', per_row=1,
...                     label=['Background', 'Pt core', 'Fe shell'],
...                     scalebar=[0], scalebar_color='white',
...                     padding={'top': 0.95, 'bottom': 0.05,
...                               'left': 0.05, 'right': 0.78})
```

Note: Because it does not make sense, it is not allowed to use a list or other iterable type for the `cmap` argument together with `'single'` for the `colorbar` argument. Such an input will cause a warning and instead set the `colorbar` argument to `None`.

It is also possible to plot multiple images overlaid on the same figure by passing the argument `overlay=True` to the `plot_images()` function. This should only be done when images have the same scale (eg. for elemental maps from

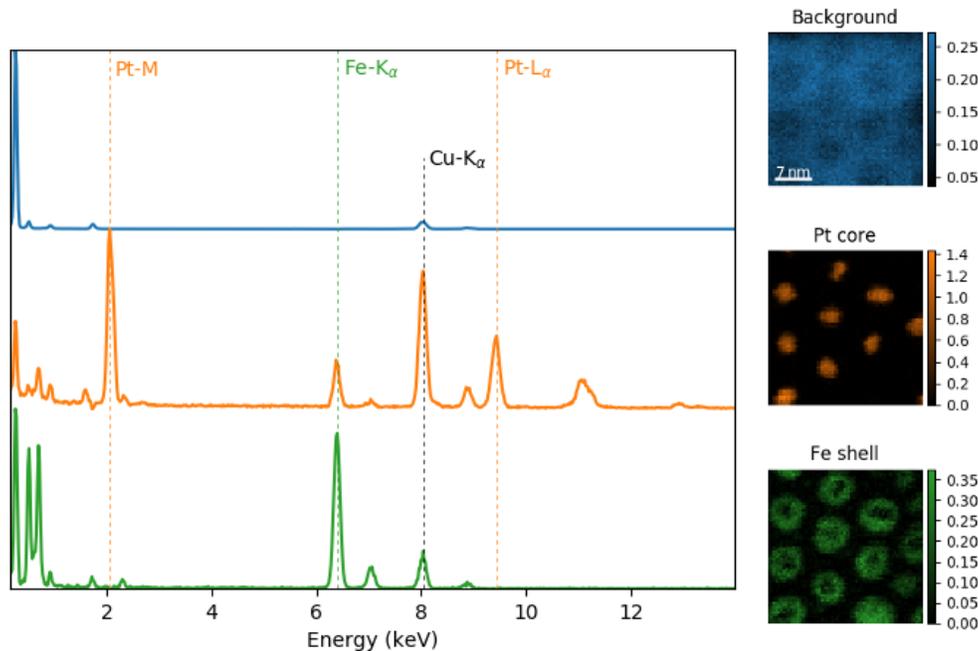


Fig. 21: Using `plot_images()` with `cmap='mpl_colors'` together with `plot_spectra()` to visualize the output of a non-negative matrix factorization of the EDS data.

the same dataset). Using the same data as above, the Fe and Pt signals can be plotted using different colours. Any color can be input via matplotlib color characters or hex values.

```
>>> si_EDS = hs.load("core_shell.hdf5")
>>> im = si_EDS.get_lines_intensity()
>>> hs.plot.plot_images(im, scalebar='all', overlay=True, subtitle=False,
...                    axes_decor='off')
```

8.6.2 Plotting several spectra

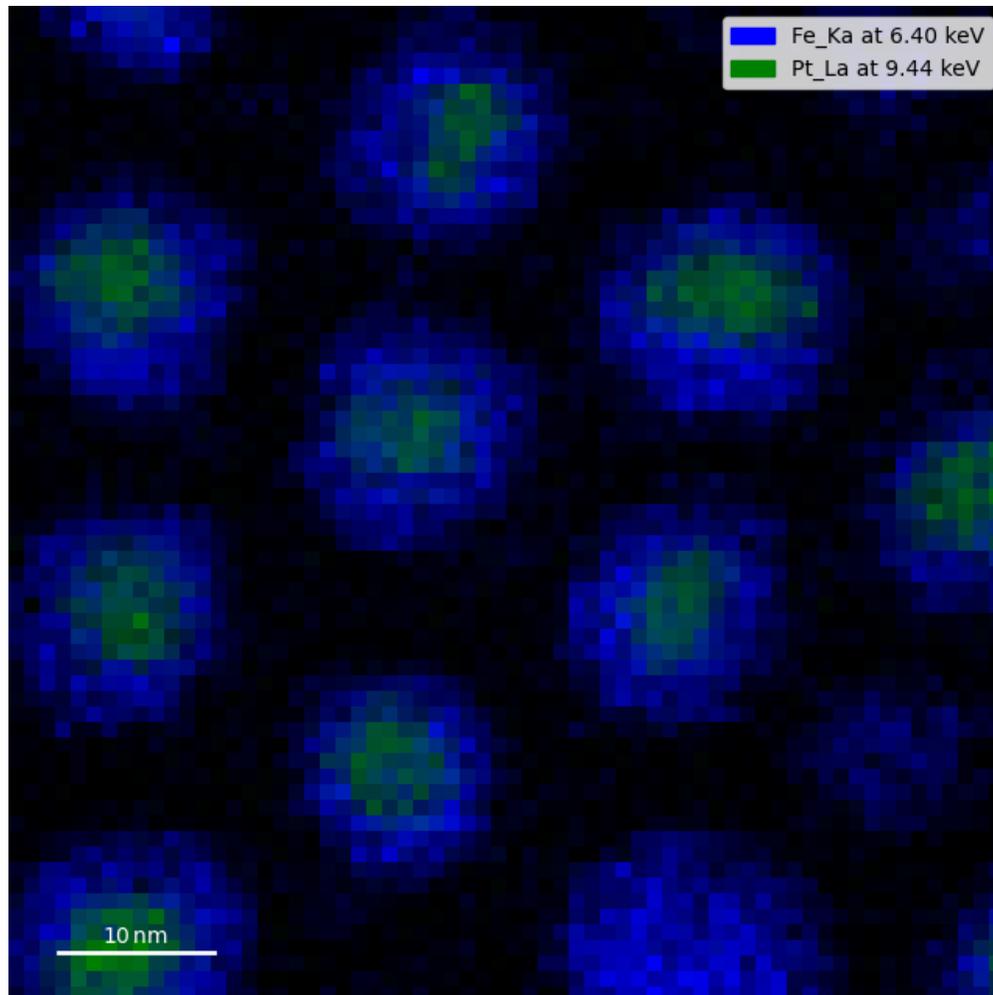
`plot_spectra()` is used to plot several spectra in the same figure. It supports different styles, the default being “overlap”.

New in version 1.5: Add support for plotting `BaseSignal` with navigation dimension 1 and signal dimension 0.

In the following example we create a list of 9 single spectra (gaussian functions with different sigma values) and plot them in the same figure using `plot_spectra()`. Note that, in this case, the legend labels are taken from the individual spectrum titles. By clicking on the legended line, a spectrum can be toggled on and off.

```
>>> s = hs.signals.Signal1D(np.zeros((200)))
>>> s.axes_manager[0].offset = -10
>>> s.axes_manager[0].scale = 0.1
>>> m = s.create_model()
>>> g = hs.model.components1D.Gaussian()
>>> m.append(g)
>>> gaussians = []
>>> labels = []
```

(continues on next page)



(continued from previous page)

```

>>> for sigma in range(1, 10):
...     g.sigma.value = sigma
...     gs = m.as_signal()
...     gs.metadata.General.title = "sigma=%i" % sigma
...     gaussians.append(gs)
...
>>> hs.plot.plot_spectra(gaussians, legend='auto')
<Axes: xlabel='<undefined> (<undefined>)', ylabel='Intensity'>

```

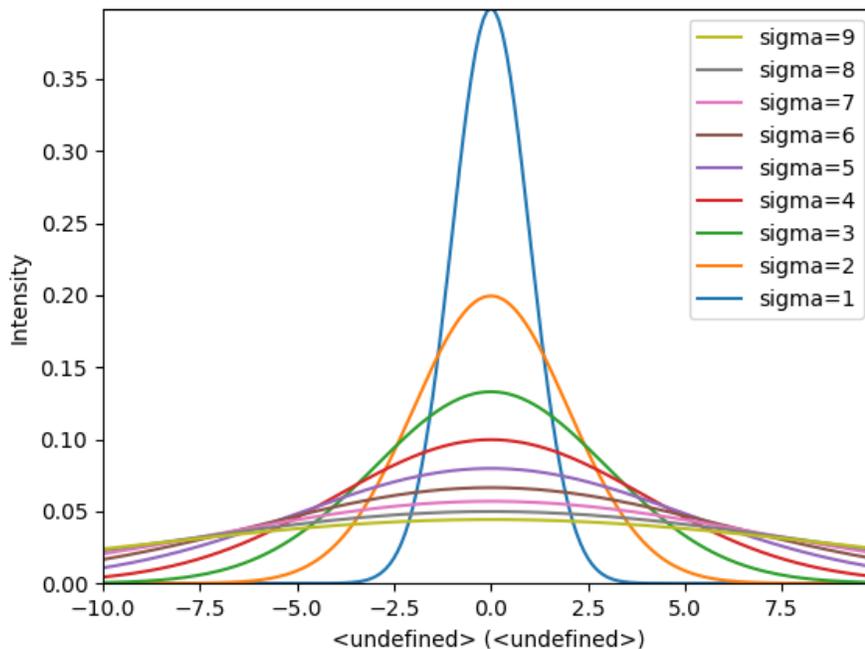


Fig. 22: Figure generated by `plot_spectra()` using the `overlap` style.

Another style, “cascade”, can be useful when “overlap” results in a plot that is too cluttered e.g. to visualize changes in EELS fine structure over a line scan. The following example shows how to plot a cascade style figure from a spectrum, and save it in a file:

```

>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> cascade_plot = hs.plot.plot_spectra(s, style='cascade')
>>> cascade_plot.figure.savefig("cascade_plot.png")

```

The “cascade” style has a `padding` option. The default value, 1, keeps the individual plots from overlapping. However in most cases a lower padding value can be used, to get tighter plots.

Using the `color` argument one can assign a color to all the spectra, or specific colors for each spectrum. In the same way, one can also assign the line style and provide the legend labels:

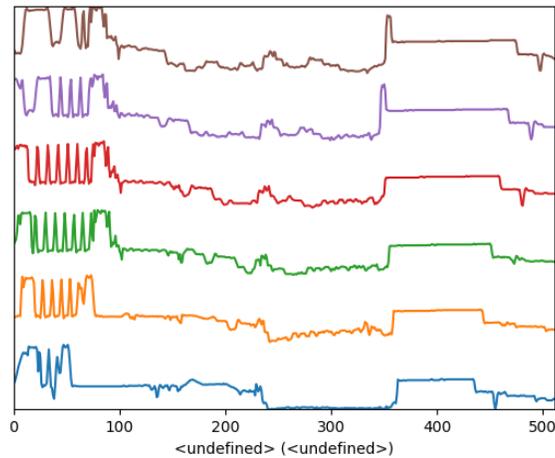


Fig. 23: Figure generated by `plot_spectra()` using the `cascade` style.

```
>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> color_list = ['red', 'red', 'blue', 'blue', 'red', 'red']
>>> linestyle_list = ['-', '--', '-.', ':', '-']
>>> hs.plot.plot_spectra(s, style='cascade', color=color_list,
...                     linestyle=linestyle_list, legend='auto')
<Axes: xlabel='<undefined> (<undefined>)'>
```

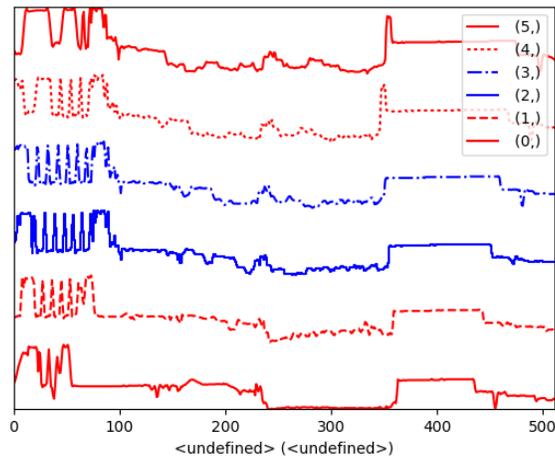


Fig. 24: Customising the line colors in `plot_spectra()`.

A simple extension of this functionality is to customize the colormap that is used to generate the list of colors. Using a list comprehension, one can generate a list of colors that follows a certain colormap:

```
>>> import scipy
>>> fig, axarr = plt.subplots(1,2)
```

(continues on next page)

(continued from previous page)

```

>>> s1 = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> s2 = hs.signals.Signal1D(scipy.datasets.ascent()[200:260:10])
>>> hs.plot.plot_spectra(s1,
...                       style='cascade',
...                       color=[plt.cm.RdBu(i/float(len(s1)-1))
...                               for i in range(len(s1))],
...                       ax=axarr[0],
...                       fig=fig)
<Axes: xlabel='<undefined> (<undefined>)'>
>>> hs.plot.plot_spectra(s2,
...                       style='cascade',
...                       color=[plt.cm.summer(i/float(len(s1)-1))
...                               for i in range(len(s1))],
...                       ax=axarr[1],
...                       fig=fig)
<Axes: xlabel='<undefined> (<undefined>)'>
>>> axarr[0].set_xlabel('RdBu (colormap)')
Text(0.5, 0, 'RdBu (colormap)')
>>> axarr[1].set_xlabel('summer (colormap)')
Text(0.5, 0, 'summer (colormap)')

```

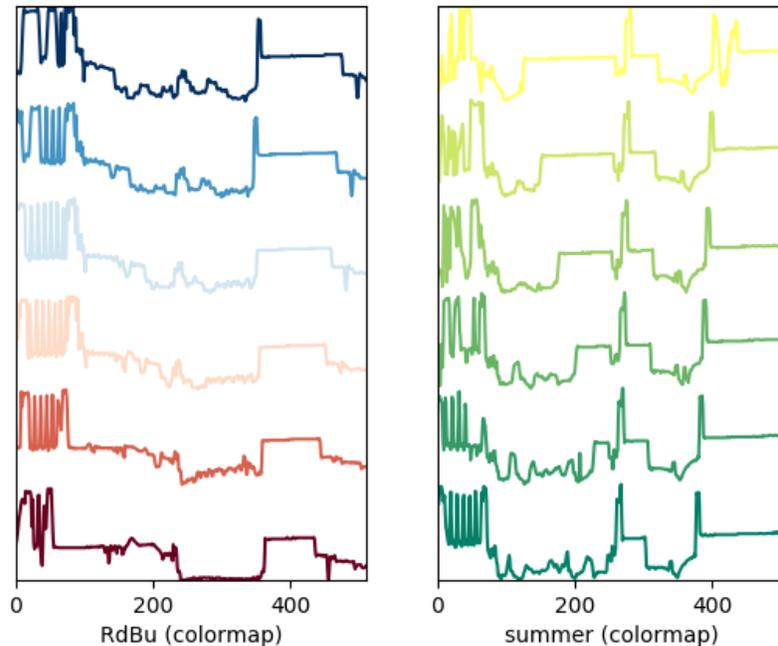


Fig. 25: Customising the line colors in `plot_spectra()` using a colormap.

There are also two other styles, “heatmap” and “mosaic”:

```

>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])

```

(continues on next page)

(continued from previous page)

```
>>> hs.plot.plot_spectra(s, style='heatmap')
<Axes: title={'center': 'Signal'}, xlabel='Unnamed 0th axis', ylabel='Spectra'>
```

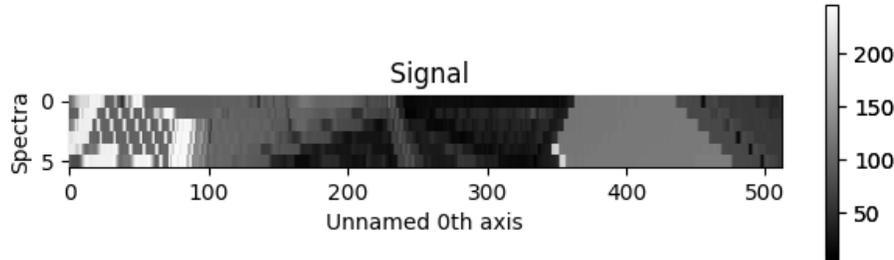


Fig. 26: Figure generated by `plot_spectra()` using the `heatmap` style.

```
>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:120:10])
>>> hs.plot.plot_spectra(s, style='mosaic')
array([<Axes: ylabel='Intensity'>,
       <Axes: xlabel='<undefined> (<undefined>)', ylabel='Intensity'>],
      dtype=object)
```

For the “heatmap” style, different `matplotlib` color schemes can be used:

```
>>> import matplotlib.cm
>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:120:10])
>>> ax = hs.plot.plot_spectra(s, style="heatmap")
>>> ax.images[0].set_cmap(matplotlib.cm.plasma)
```

Any parameter that can be passed to `matplotlib.pyplot.figure` can also be used with `plot_spectra()` to allow further customization (when using the “overlap”, “cascade”, or “mosaic” styles). In the following example, `dpi`, `facecolor`, `frameon`, and `num` are all parameters that are passed directly to `matplotlib.pyplot.figure` as keyword arguments:

```
>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> legendtext = ['Plot 0', 'Plot 1', 'Plot 2', 'Plot 3',
...              'Plot 4', 'Plot 5']
>>> cascade_plot = hs.plot.plot_spectra(
...     s, style='cascade', legend=legendtext, dpi=60,
...     facecolor='lightblue', frameon=True, num=5)
>>> cascade_plot.set_xlabel("X-axis")
Text(0.5, 0, 'X-axis')
>>> cascade_plot.set_ylabel("Y-axis")
Text(0, 0.5, 'Y-axis')
>>> cascade_plot.set_title("Cascade plot")
Text(0.5, 1.0, 'Cascade plot')
```

The function returns a `matplotlib` `ax` object, which can be used to customize the figure:

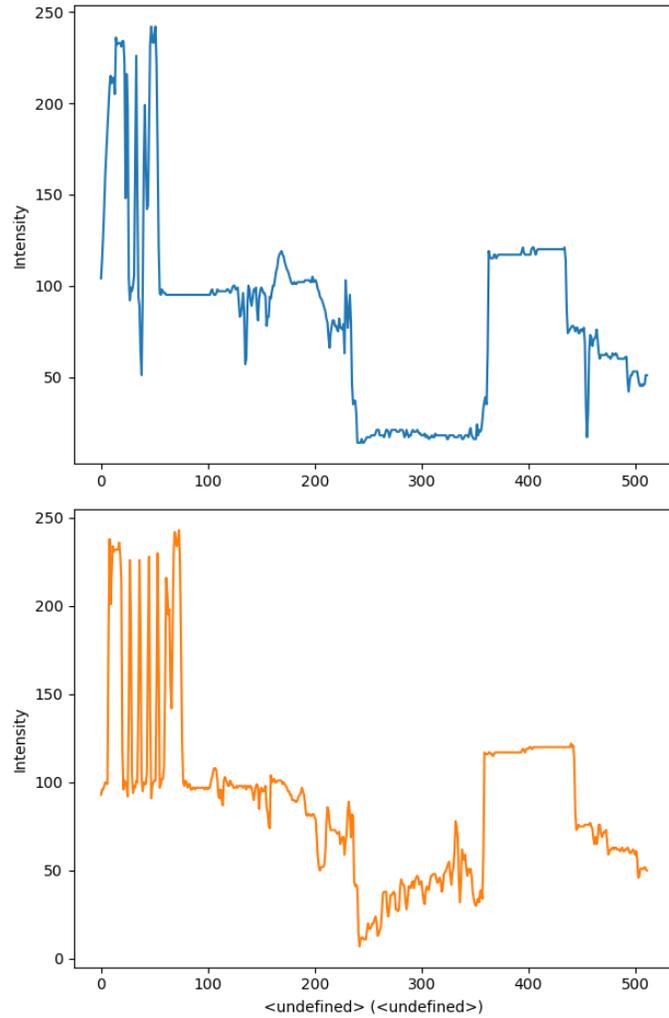


Fig. 27: Figure generated by `plot_spectra()` using the *mosaic* style.

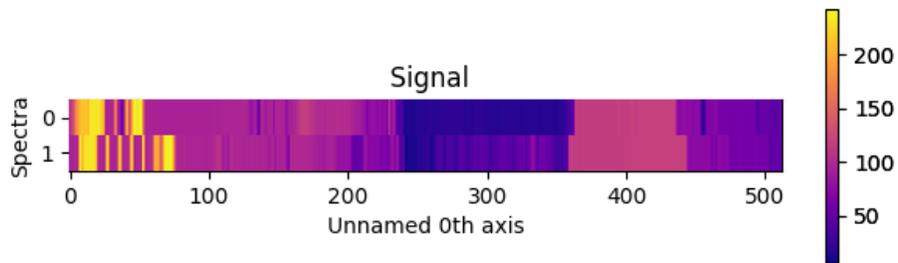


Fig. 28: Figure generated by `plot_spectra()` using the *heatmap* style showing how to customise the color map.

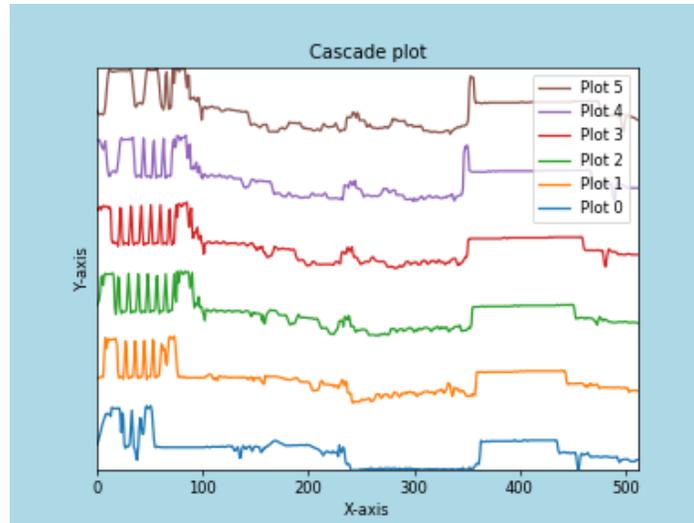


Fig. 29: Customising the figure with keyword arguments.

```

>>> import scipy
>>> s = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> cascade_plot = hs.plot.plot_spectra(s)
>>> cascade_plot.set_xlabel("An axis")
Text(0.5, 0, 'An axis')
>>> cascade_plot.set_ylabel("Another axis")
Text(0, 0.5, 'Another axis')
>>> cascade_plot.set_title("A title!")
Text(0.5, 1.0, 'A title!')

```

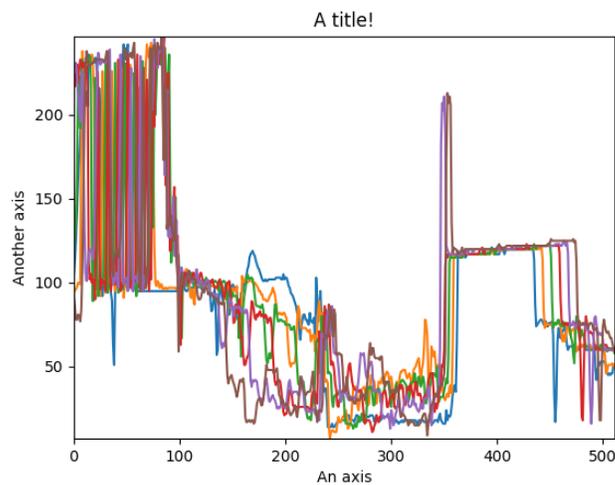


Fig. 30: Customising the figure by setting the matplotlib axes properties.

A matplotlib ax and fig object can also be specified, which can be used to put several subplots in the same figure. This will only work for “cascade” and “overlap” styles:

```

>>> import scipy
>>> fig, axarr = plt.subplots(1,2)
>>> s1 = hs.signals.Signal1D(scipy.datasets.ascent()[100:160:10])
>>> s2 = hs.signals.Signal1D(scipy.datasets.ascent()[200:260:10])
>>> hs.plot.plot_spectra(s1, style='cascade',
...                      color='blue', ax=axarr[0], fig=fig)
<Axes: xlabel='<undefined> (<undefined>)'>
>>> hs.plot.plot_spectra(s2, style='cascade',
...                      color='red', ax=axarr[1], fig=fig)
<Axes: xlabel='<undefined> (<undefined>)'>

```

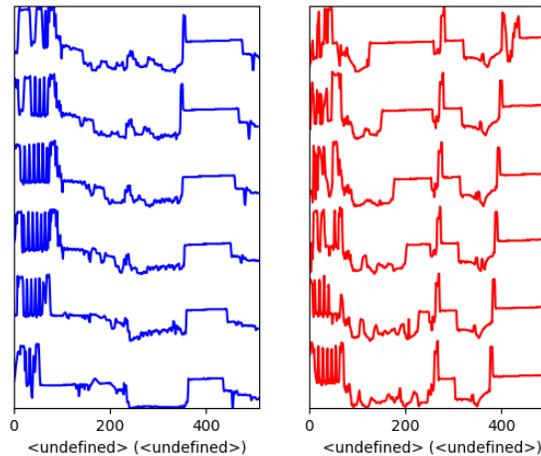


Fig. 31: Plotting on existing matplotlib axes.

Plotting several signals

`plot_signals()` is used to plot several signals at the same time. By default the navigation position of the signals will be synced, and the signals must have the same dimensions. To plot two spectra at the same time:

```

>>> import scipy
>>> s1 = hs.signals.Signal1D(scipy.datasets.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> hs.plot.plot_signals([s1, s2])

```

The navigator can be specified by using the `navigator` argument, where the different options are “auto”, None, “spectrum”, “slider” or Signal. For more details about the different navigators, see [the navigator options](#). To specify the navigator:

```

>>> import scipy
>>> s1 = hs.signals.Signal1D(scipy.datasets.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> hs.plot.plot_signals([s1, s2], navigator="slider")

```

Navigators can also be set differently for different plots using the `navigator_list` argument. Where the `navigator_list` be the same length as the number of signals plotted, and only contain valid navigator options. For example:

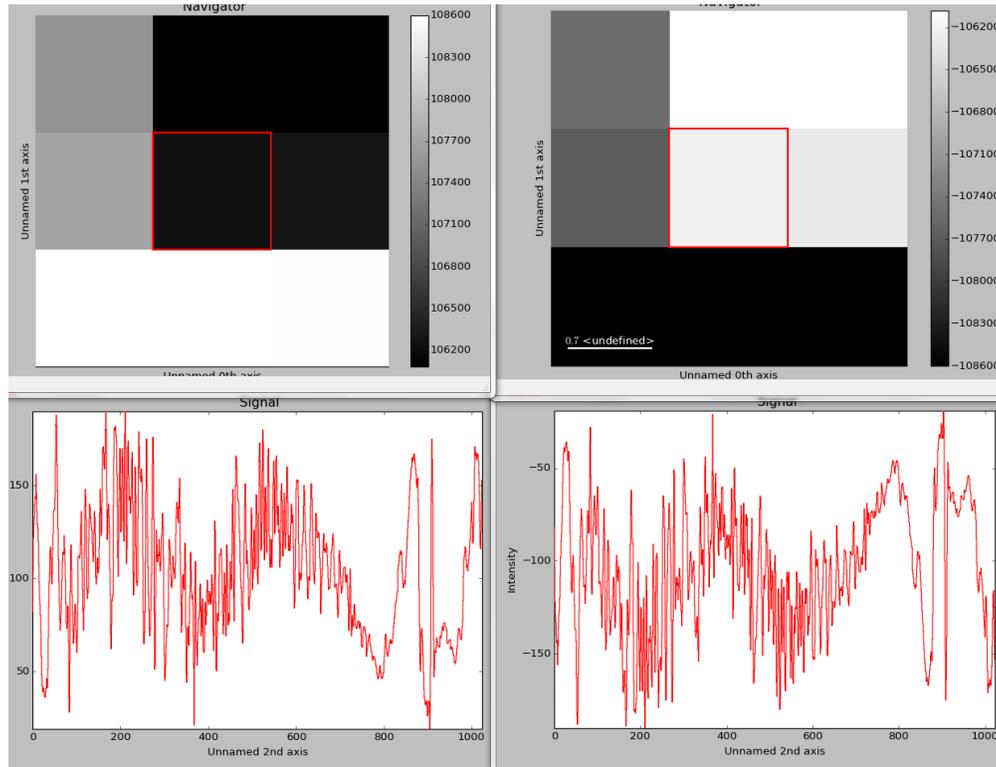


Fig. 32: The `plot_signals()` plots several signals with optional synchronized navigation.

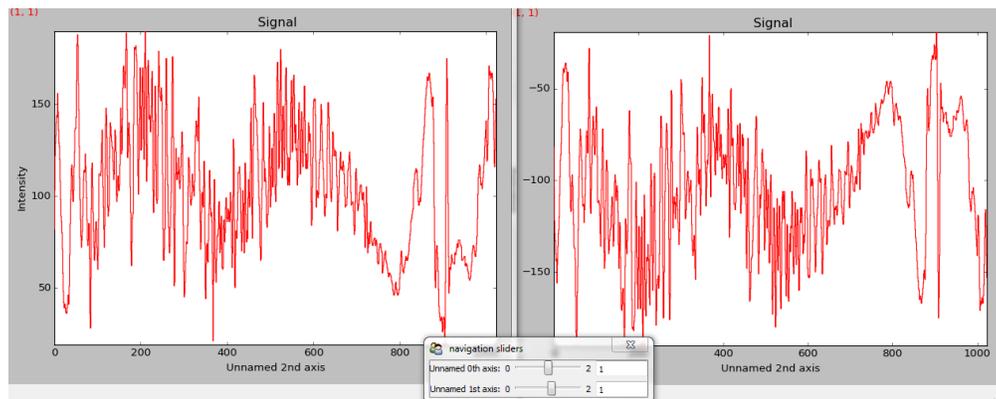


Fig. 33: Customising the navigator in `plot_signals()`.

```

>>> import scipy
>>> s1 = hs.signals.Signal1D(scipy.datasets.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> s3 = hs.signals.Signal1D(np.linspace(0,9,9).reshape([3,3]))
>>> hs.plot.plot_signals([s1, s2], navigator_list=["slider", s3])

```

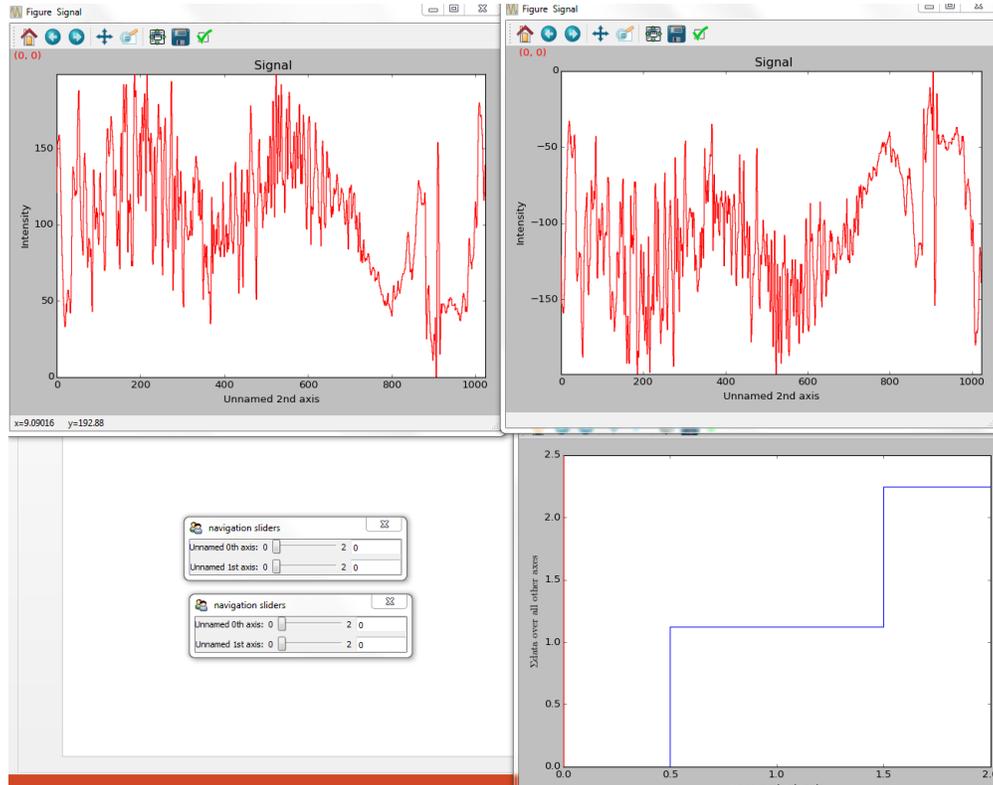


Fig. 34: Customising the navigator in `plot_signals()` by providing a navigator list.

Several signals can also be plotted without syncing the navigation by using `sync=False`. The `navigator_list` can still be used to specify a navigator for each plot:

```

>>> import scipy
>>> s1 = hs.signals.Signal1D(scipy.datasets.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> hs.plot.plot_signals([s1, s2], sync=False, navigator_list=["slider", "slider"])

```

8.7 Markers

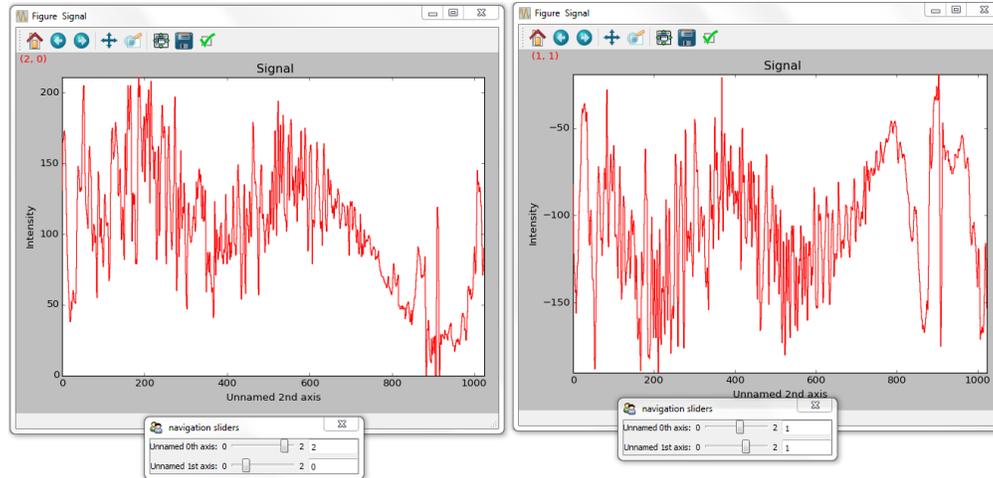
HyperSpy provides an easy access the collections classes of matplotlib. These markers provide powerful ways to annotate high dimensional datasets easily.

```

>>> import scipy
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> m = hs.plot.markers.Rectangles(
...     offsets=[[275, 250],], widths=[250,],

```

(continues on next page)

Fig. 35: Disabling synchronised navigation in `plot_signals()`.

(continued from previous page)

```
... heights=[300],color="red", facecolor="none")
>>> im.add_marker(m)
```

By providing an array of positions, the marker can also change position when navigating the signal. In the following example, the local maxima are displayed for each R, G and B channel of a colour image.

```
>>> from skimage.feature import peak_local_max
>>> import scipy
>>> ims = hs.signals.BaseSignal(scipy.datasets.face()).as_signal2D([1,2])
>>> index = ims.map(peak_local_max,min_distance=100,
...                 num_peaks=4, inplace=False, ragged=True)
>>> m = hs.plot.markers.Points.from_signal(index, color='red')
>>> ims.add_marker(m)
```

Markers can be added to the navigator as well. In the following example, each slice of a 2D spectrum is tagged with a text marker on the signal plot. Each slice is indicated with the same text on the navigator.

```
>>> import numpy as np
>>> s = hs.signals.Signal1D(np.arange(100).reshape([10,10]))
>>> s.plot(navigator='spectrum')
>>> offsets = [[i, s.sum(-1).data[i]+5] for i in range(s.axes_manager.shape[0])]
>>> text = 'abcdefghij'
>>> m = hs.plot.markers.Texts(offsets=offsets, texts=[*text], verticalalignment="bottom")
>>> s.add_marker(m, plot_on_signal=False)
>>> offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> texts = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> x = 4
>>> for i in range(10):
...     offsets[i] = [[x, int(s.inav[i].isig[x].data)],]
...     texts[i] = np.array([text[i],])
>>> m_sig = hs.plot.markers.Texts(offsets=offsets, texts=texts, verticalalignment="bottom",
...                               ↵)
>>> s.add_marker(m_sig)
```

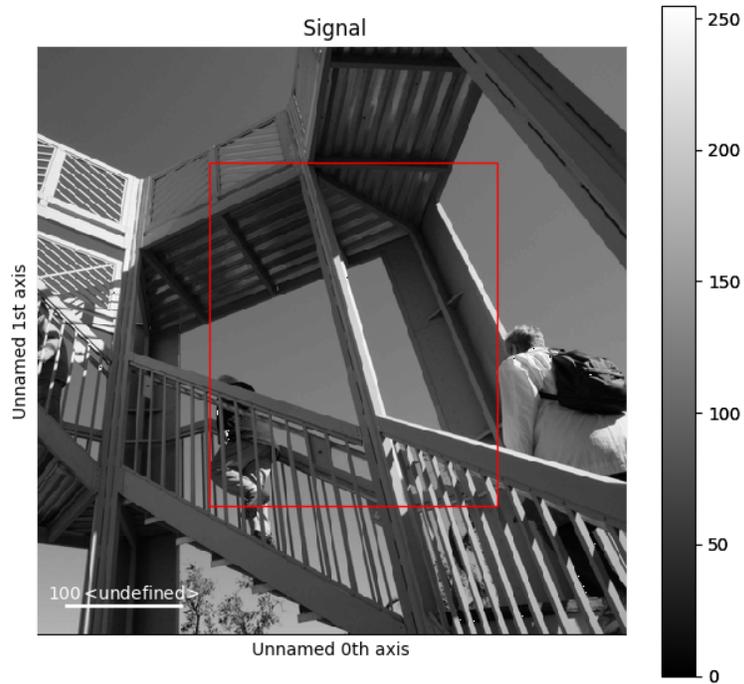


Fig. 36: Rectangle static marker.

Fig. 37: Point markers in image.

Fig. 38: Multi-dimensional markers.

8.7.1 Permanent markers

New in version 1.2: Permanent markers.

These markers can also be permanently added to a signal, which is saved in `metadata.Markers`:

```
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.plot.markers.Points(offsets = [[5,9]], sizes=1, units="xy")
>>> s.add_marker(marker, permanent=True)
>>> s.metadata.Markers
└─ Points = <Points (Points), length: 1>
>>> s.plot()
```

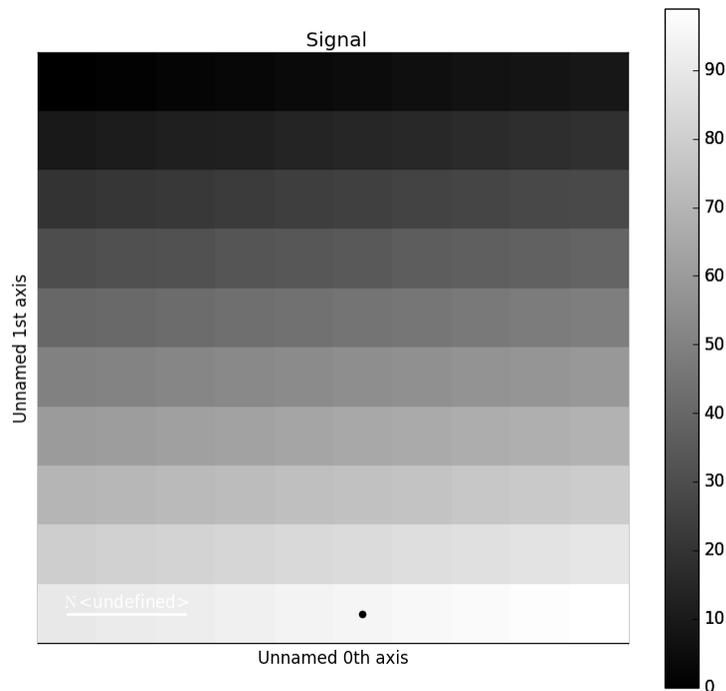


Fig. 39: Plotting with permanent markers.

Markers can be removed by deleting them from the metadata

```
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.plot.markers.Points(offsets = [[5,9]], sizes=1)
>>> s.add_marker(marker, permanent=True)
>>> s.metadata.Markers
└─ Points = <Points (Points), length: 1>
>>> del s.metadata.Markers.Points
>>> s.metadata.Markers # Returns nothing
```

To suppress plotting of permanent markers, use `plot_markers=False` when calling `s.plot`:

```
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.plot.markers.Points(offsets=[[5,9]], sizes=1, units="xy")
>>> s.add_marker(marker, permanent=True, plot_marker=False)
>>> s.plot(plot_markers=False)
```

If the signal has a navigation dimension, the markers can be made to change as a function of the navigation index by passing in kwargs with dtype=object. For a signal with 1 navigation axis:

```
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> marker_pos = [[5,9], [1,8], [2,1]]
>>> for i,m in zip(np.ndindex(3), marker_pos):
...     offsets[i] = m
>>> marker = hs.plot.markers.Points(offsets=offsets, color="red", sizes=10)
>>> s.add_marker(marker, permanent=True)
```

Plotting with markers that change with the navigation index.

Or for a signal with 2 navigation axes:

```
>>> s = hs.signals.Signal2D(np.arange(400).reshape(2, 2, 10, 10))
>>> marker_pos = np.array([[5,1], [1,2]],[2,9],[6,8]])
>>> offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> for i in np.ndindex(s.axes_manager.navigation_shape):
...     offsets[i] = [marker_pos[i],]
>>> marker = hs.plot.markers.Points(offsets=offsets, sizes=10)
>>> s.add_marker(marker, permanent=True)
```

Fig. 40: Plotting with markers that change with the two-dimensional navigation index.

This can be extended to 4 (or more) navigation dimensions:

```
>>> s = hs.signals.Signal2D(np.arange(1600).reshape(2, 2, 2, 2, 10, 10))
>>> x = np.arange(16).reshape(2, 2, 2, 2)
>>> y = np.arange(16).reshape(2, 2, 2, 2)
>>> offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> for i in np.ndindex(s.axes_manager.navigation_shape):
...     offsets[i] = [[x[i],y[i]],]
>>> marker = hs.plot.markers.Points(offsets=offsets, color='red', sizes=10)
>>> s.add_marker(marker, permanent=True)
```

You can add a couple of different types of markers at the same time.

```
>>> import hyperspy.api as hs
>>> import numpy as np
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> markers = []
>>> v_line_pos = np.empty(3, dtype=object)
>>> point_offsets = np.empty(3, dtype=object)
>>> text_offsets = np.empty(3, dtype=object)
>>> h_line_pos = np.empty(3, dtype=object)
>>> random_colors = np.empty(3, dtype=object)
>>> num=200
>>> for i in range(3):
...     v_line_pos[i] = np.random.rand(num)*10
...     h_line_pos[i] = np.random.rand(num)*10
...     point_offsets[i] = np.random.rand(num,2)*10
```

(continues on next page)

(continued from previous page)

```

...     text_offsets[i] = np.random.rand(num,2)*10
...     random_colors = np.random.rand(num,3)
>>> v_marker = hs.plot.markers.VerticalLines(offsets=v_line_pos, color=random_colors)
>>> h_marker = hs.plot.markers.HorizontalLines(offsets=h_line_pos, color=random_colors)
>>> p_marker = hs.plot.markers.Points(offsets=point_offsets, color=random_colors,
↳ sizes=(.1,))
>>> t_marker = hs.plot.markers.Texts(offsets=text_offsets, texts=["sometext", ])
>>> s.add_marker([v_marker,h_marker, p_marker, t_marker], permanent=True)

```

Fig. 41: Plotting many types of markers with an iterable so they change with the navigation index.

Permanent markers are stored in the HDF5 file if the signal is saved:

```

>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.plot.markers.Points([[2, 1]], color='red')
>>> s.add_marker(marker, plot_marker=False, permanent=True)
>>> s.metadata.Markers
└─ Points = <Points (Points), length: 1>
>>> s.save("storing_marker.hspy")
>>> s1 = hs.load("storing_marker.hspy")
>>> s1.metadata.Markers
└─ Points = <Points (Points), length: 1>

```

8.7.2 Supported markers

The markers currently supported in HyperSpy are:

Table 3: List of supported markers, their signature and their corresponding matplotlib objects.

HyperSpy Markers	Signature	Matplotlib Collection
<i>Arrows</i>	offsets, U, V, C, **kwargs	matplotlib.quiver.Quiver
<i>Circles</i>	offsets, sizes, **kwargs	matplotlib.collections.CircleCollection
<i>Ellipses</i>	offsets, widths, heights, **kwargs	matplotlib.collections.EllipseCollection
<i>HorizontalLines</i>	offsets, **kwargs	matplotlib.collections.LineCollection
<i>Lines</i>	segments, **kwargs	matplotlib.collections.LineCollection
<i>Markers</i>	offsets, **kwargs	
<i>Points</i>	offsets, **kwargs	matplotlib.collections.CircleCollection
<i>Polygons</i>	verts, **kwargs	matplotlib.collections.PolyCollection
<i>Rectangles</i>	offsets, widths, heights, **kwargs	Custom RectangleCollection
<i>Squares</i>	offsets, widths, **kwargs	Custom SquareCollection
<i>Texts</i>	offsets, texts, **kwargs	Custom TextCollection
<i>VerticalLines</i>	offsets, **kwargs	matplotlib.collections.LineCollection

8.7.3 Marker properties

The optional parameters (**kwargs, keyword arguments) can be used for extra parameters used for each matplotlib collection. Any parameter which can be set using the `matplotlib.collections.Collection.set()` method can be used as an iterating parameter with respect to the navigation index by passing in a numpy array with `dtype=object`. Otherwise to set the parameter globally the kwarg can directly be passed.

Additionally, if some **kwargs are shorter in length to some other parameter it will be cycled such that

```
>>> prop[i % len(prop)]
```

where `i` is the `i`th element of the collection.

8.7.4 Extra information about Markers

New in version 2.0: Marker Collections for faster plotting of many markers

Hyperspy's `Markers` class and its subclasses extends the capabilities of the `matplotlib.collections.Collection` class and subclasses. Primarily it allows dynamic markers to be initialized by passing key word arguments with `dtype=object`. Those attributes are then updated with the plot as you navigate through the plot.

In most cases the `offsets` kwarg is used to map some marker to multiple positions in the plot. For example we can define a plot of Ellipses using:

```
>>> import numpy as np
>>> import hyperspy.api as hs
>>> hs.plot.markers.Ellipses(heights=(.4,), widths=(1,),
...                          angles=(10,), offsets=np.array([[0,0], [1,1]]))
<Ellipses, length: 2>
```

Alternatively, if we want to make ellipses with different heights and widths we can pass multiple values to `heights`, `widths` and `angles`. In general these properties will be applied such that `prop[i % len(prop)]` so passing `heights=(.1, .2, .3)` will result in the ellipse at `offsets[0]` with a height of 0.1 the ellipse at `offsets[1]` with a height of 0.1, ellipse at `offsets[2]` has a height of 0.3 and the ellipse at `offsets[3]` has a height of 0.1 and so on.

For attributes which we want to be dynamic and change with the navigation coordinates we can pass those values as an array with `dtype=object`. Each of those values will be set as the index changes.

Note: Only kwargs which can be passed to `matplotlib.collections.Collection.set()` can be dynamic.

If we want to plot a series of points, we can use the following code, in this case both the `sizes` and `offsets` kwargs are dynamic and change with each index.

```
>>> import numpy as np
>>> import hyperspy.api as hs
>>> data = np.empty((2,2), dtype=object)
>>> sizes = np.empty((2,2), dtype=object)
>>> for i, ind in enumerate(np.ndindex((2,2))):
...     data[ind] = np.random.rand(i+1,2)*3 # dynamic positions
...     sizes[ind] = [(i+1)/10,] # dynamic sizes
>>> m = hs.plot.markers.Points(sizes=sizes, offsets=data, color="r", units="xy")
>>> s = hs.signals.Signal2D(np.zeros((2,2,4,4)))
>>> s.plot()
>>> s.add_marker(m)
```

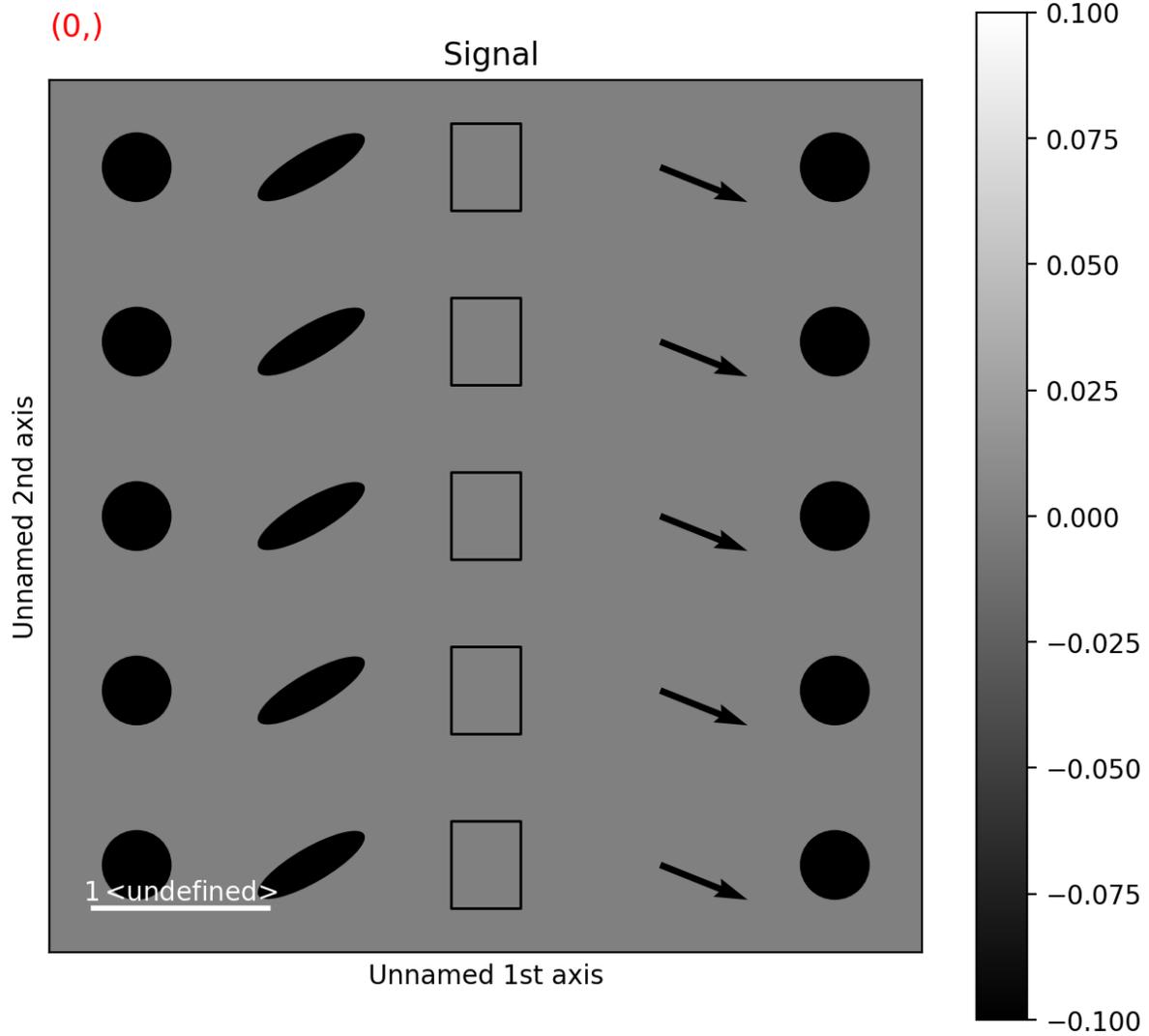
The *Markers* also has a class method `from_signal()` which can be used to create a set of markers from the output of some map function. In this case `signal.data` is mapped to some key and used to initialize a *Markers* object. If the signal has the attribute `signal.metadata.Peaks.signal_axes` and `convert_units = True` then the values will be converted to the proper units before creating the *Markers* object.

Note: For kwargs like `size`, `height`, etc. the scale and the units of the x axis are used to plot.

Let's consider how plotting a bunch of different collections might look:

```
>>> import hyperspy.api as hs
>>> import numpy as np

>>> collections = [hs.plot.markers.Points,
...               hs.plot.markers.Ellipses,
...               hs.plot.markers.Rectangles,
...               hs.plot.markers.Arrows,
...               hs.plot.markers.Circles,
...               ]
>>> num_col = len(collections)
>>> offsets = [np.stack([np.ones(num_col)*i, np.arange(num_col)], axis=1) for i in
↳range(len(collections))]
>>> kwargs = [{"sizes":(.4,),"facecolor":"black"},
...           {"widths":(.2,),"heights":(.7,),"angles":(60,),"facecolor":"black"},
...           {"widths":(.4,),"heights":(.5,),"facecolor":"none","edgecolor":"black"},
...           {"U":(.5,),"V":(.2,),"facecolor":"black"},
...           {"sizes":(.4,),"facecolor":"black"},]
>>> for k, o, c in zip(kwargs, offsets, collections):
...     k["offsets"] = o
>>> collections = [C(**k) for k,C in zip(kwargs, collections)]
>>> s = hs.signals.Signal2D(np.zeros((2, num_col, num_col)))
>>> s.plot()
>>> s.add_marker(collections)
```



MACHINE LEARNING

HyperSpy provides easy access to several “machine learning” algorithms that can be useful when analysing multi-dimensional data. In particular, decomposition algorithms, such as principal component analysis (PCA), or blind source separation (BSS) algorithms, such as independent component analysis (ICA), are available through the methods described in this section.

Hint: HyperSpy will decompose a dataset, X , into two new datasets: one with the dimension of the signal space known as **factors** (A), and the other with the dimension of the navigation space known as **loadings** (B), such that $X = AB^T$.

For some of the algorithms listed below, the decomposition results in an *approximation* of the dataset, i.e. $X \approx AB^T$.

9.1 Decomposition

Decomposition techniques are most commonly used as a means of noise reduction (or *denoising*) and dimensionality reduction. To apply a decomposition to your dataset, run the `decomposition()` method, for example:

```
>>> s = hs.signals.Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition()
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=None
  centre=None

>>> # Load data from a file, then decompose
>>> s = hs.load("my_file.hspy")
>>> s.decomposition()
```

Note: The signal `s` must be multi-dimensional, i.e. `s.axes_manager.navigation_size > 1`

One of the most popular uses of `decomposition()` is data denoising. This is achieved by using a limited set of components to make a model of the original dataset, omitting the less significant components that ideally contain only noise.

To reconstruct your denoised or reduced model, run the `get_decomposition_model()` method. For example:

```

>>> # Use all components to reconstruct the model
>>> sc = s.get_decomposition_model()

>>> # Use first 3 components to reconstruct the model
>>> sc = s.get_decomposition_model(3)

>>> # Use components [0, 2] to reconstruct the model
>>> sc = s.get_decomposition_model([0, 2])

```

Sometimes, it is useful to examine the residuals between your original data and the decomposition model. You can easily calculate and display the residuals, since `get_decomposition_model()` returns a new object, which in the example above we have called `sc`:

```
>>> (s - sc).plot()
```

You can perform operations on this new object `sc` later. It is a copy of the original `s` object, except that the data has been replaced by the model constructed using the chosen components.

If you provide the `output_dimension` argument, which takes an integer value, the decomposition algorithm attempts to find the best approximation for the dataset X with only a limited set of factors A and loadings B , such that $X \approx AB^T$.

```
>>> s.decomposition(output_dimension=3)
```

Some of the algorithms described below require `output_dimension` to be provided.

9.1.1 Available algorithms

HyperSpy implements a number of decomposition algorithms via the `algorithm` argument. The table below lists the algorithms that are currently available, and includes links to the appropriate documentation for more information on each one.

Note: Choosing which algorithm to use is likely to depend heavily on the nature of your dataset and the type of analysis you are trying to perform. We discuss some of the reasons for choosing one algorithm over another below, but would encourage you to do your own research as well. The [scikit-learn documentation](#) is a very good starting point.

Table 1: Available decomposition algorithms in HyperSpy

Algorithm	Method
“SVD” (default)	<code>svd_pca()</code>
“MLPCA”	<code>mlpca()</code>
“sklearn_pca”	<code>sklearn.decomposition.PCA</code>
“NMF”	<code>sklearn.decomposition.NMF</code>
“sparse_pca”	<code>sklearn.decomposition.SparsePCA</code>
“mini_batch_sparse_pca”	<code>sklearn.decomposition.MinibatchSparsePCA</code>
“RPCA”	<code>rpca_godec()</code>
“ORPCA”	<code>ORPCA</code>
“ORNMF”	<code>ORNMF</code>
custom object	An object implementing <code>fit()</code> and <code>transform()</code> methods

9.1.2 Singular value decomposition (SVD)

The default algorithm in HyperSpy is "SVD", which uses an approach called "singular value decomposition" to decompose the data in the form $X = U\Sigma V^T$. The factors are given by $U\Sigma$, and the loadings are given by V^T . For more information, please read the method documentation for `svd_pca()`.

```
>>> s = hs.signals.Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition()
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=None
  centre=None
```

Note: In some fields, including electron microscopy, this approach of applying an SVD directly to the data X is often called PCA (*see below*).

However, in the classical definition of PCA, the SVD should be applied to data that has first been "centered" by subtracting the mean, i.e. $SVD(X - \bar{X})$.

The "SVD" algorithm in HyperSpy **does not** apply this centering step by default. As a result, you may observe differences between the output of the "SVD" algorithm and, for example, `sklearn.decomposition.PCA`, which **does** apply centering.

9.1.3 Principal component analysis (PCA)

One of the most popular decomposition methods is [principal component analysis](#) (PCA). To perform PCA on your dataset, run the `decomposition()` method with any of following arguments.

If you have `scikit-learn` installed:

```
>>> s.decomposition(algorithm="sklearn_pca")
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=sklearn_pca
  output_dimension=None
  centre=None
scikit-learn estimator:
PCA()
```

You can also turn on centering with the default "SVD" algorithm via the "centre" argument:

```
# Subtract the mean along the navigation axis
>>> s.decomposition(algorithm="SVD", centre="navigation")
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=None
  centre=navigation

# Subtract the mean along the signal axis
>>> s.decomposition(algorithm="SVD", centre="signal")
```

(continues on next page)

(continued from previous page)

```
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=None
  centre=signal
```

You can also use `sklearn.decomposition.PCA` directly:

```
>>> from sklearn.decomposition import PCA
>>> s.decomposition(algorithm=PCA())
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=PCA()
  output_dimension=None
  centre=None
scikit-learn estimator:
PCA()
```

9.1.4 Poissonian noise

Most of the standard decomposition algorithms assume that the noise of the data follows a Gaussian distribution (also known as “homoskedastic noise”). In cases where your data is instead corrupted by Poisson noise, HyperSpy can “normalize” the data by performing a scaling operation, which can greatly enhance the result. More details about the normalization procedure can be found in [\[Keenan2004\]](#).

To apply Poissonian noise normalization to your data:

```
>>> s.decomposition(normalize_poissonian_noise=True)

>>> # Because it is the first argument we could have simply written:
>>> s.decomposition(True)
```

Warning: Poisson noise normalization cannot be used in combination with data centering using the 'centre' argument. Attempting to do so will raise an error.

9.1.5 Maximum likelihood principal component analysis (MLPCA)

Instead of applying Poisson noise normalization to your data, you can instead use an approach known as Maximum Likelihood PCA (MLPCA), which provides a more robust statistical treatment of non-Gaussian “heteroskedastic noise”.

```
>>> s.decomposition(algorithm="MLPCA")
```

For more information, please read the method documentation for `mlpca()`.

Note: You must set the `output_dimension` when using MLPCA.

9.1.6 Robust principal component analysis (RPCA)

PCA is known to be very sensitive to the presence of outliers in data. These outliers can be the result of missing or dead pixels, X-ray spikes, or very low count data. If one assumes a dataset, X , to consist of a low-rank component L corrupted by a sparse error component S , such that $X = L + S$, then Robust PCA (RPCA) can be used to recover the low-rank component for subsequent processing [Candes2011].

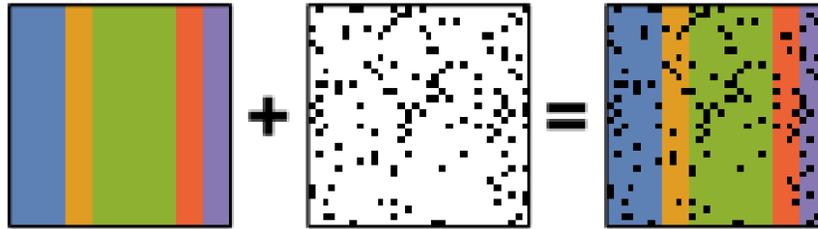


Fig. 1: Schematic diagram of the robust PCA problem, which combines a low-rank matrix with sparse errors. Robust PCA aims to decompose the matrix back into these two components.

Note: You must set the `output_dimension` when using Robust PCA.

The default RPCA algorithm is GoDec [Zhou2011]. In HyperSpy it returns the factors and loadings of L . RPCA solvers work by using regularization, in a similar manner to lasso or ridge regression, to enforce the low-rank constraint on the data. The low-rank regularization parameter, `lambda1`, defaults to `1/sqrt(n_features)`, but it is strongly recommended that you explore the behaviour of different values.

```
>>> s.decomposition(algorithm="RPCA", output_dimension=3, lambda1=0.1)
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=RPCA
  output_dimension=3
  centre=None
```

HyperSpy also implements an *online* algorithm for RPCA developed by Feng et al. [Feng2013]. This minimizes memory usage, making it suitable for large datasets, and can often be faster than the default algorithm.

```
>>> s.decomposition(algorithm="ORPCA", output_dimension=3)
```

The online RPCA implementation sets several default parameters that are usually suitable for most datasets, including the regularization parameter highlighted above. Again, it is strongly recommended that you explore the behaviour of these parameters. To further improve the convergence, you can “train” the algorithm with the first few samples of your dataset. For example, the following code will train ORPCA using the first 32 samples of the data.

```
>>> s.decomposition(algorithm="ORPCA", output_dimension=3, training_samples=32)
```

Finally, online RPCA includes two alternative methods to the default block-coordinate descent solver, which can again improve both the convergence and speed of the algorithm. These are particularly useful for very large datasets.

The methods are based on stochastic gradient descent (SGD), and take an additional parameter to set the learning rate. The learning rate dictates the size of the steps taken by the gradient descent algorithm, and setting it too large can lead to oscillations that prevent the algorithm from finding the correct minima. Usually a value between 1 and 2 works well:

```
>>> s.decomposition(algorithm="ORPCA",
...                 output_dimension=3,
...                 method="SGD",
...                 subspace_learning_rate=1.1)
```

You can also use Momentum Stochastic Gradient Descent (MomentumSGD), which typically improves the convergence properties of stochastic gradient descent. This takes the further parameter `subspace_momentum`, which should be a fraction between 0 and 1.

```
>>> s.decomposition(algorithm="ORPCA",
...                 output_dimension=3,
...                 method="MomentumSGD",
...                 subspace_learning_rate=1.1,
...                 subspace_momentum=0.5)
```

Using the "SGD" or "MomentumSGD" methods enables the subspace, i.e. the underlying low-rank component, to be tracked as it changes with each sample update. The default method instead assumes a fixed, static subspace.

9.1.7 Non-negative matrix factorization (NMF)

Another popular decomposition method is non-negative matrix factorization (NMF), which can be accessed in HyperSpy with:

```
>>> s.decomposition(algorithm="NMF")
```

Unlike PCA, NMF forces the components to be strictly non-negative, which can aid the physical interpretation of components for count data such as images, EELS or EDS. For an example of NMF in EELS processing, see [\[Nicoletti2013\]](#).

NMF takes the optional argument `output_dimension`, which determines the number of components to keep. Setting this to a small number is recommended to keep the computation time small. Often it is useful to run a PCA decomposition first and use the *scree plot* to determine a suitable value for `output_dimension`.

9.1.8 Robust non-negative matrix factorization (RNMF)

In a similar manner to the online, robust methods that complement PCA *above*, HyperSpy includes an online robust NMF method. This is based on the OPGD (Online Proximal Gradient Descent) algorithm of [\[Zhao2016\]](#).

Note: You must set the `output_dimension` when using Robust NMF.

As before, you can control the regularization applied via the parameter "lambda1":

```
>>> s.decomposition(algorithm="ORNMF", output_dimension=3, lambda1=0.1)
```

The MomentumSGD method is useful for scenarios where the subspace, i.e. the underlying low-rank component, is changing over time.

```
>>> s.decomposition(algorithm="ORNMF",
...                 output_dimension=3,
...                 method="MomentumSGD",
...                 subspace_learning_rate=1.1,
...                 subspace_momentum=0.5)
```

Both the default and MomentumSGD solvers assume an l_2 -norm minimization problem, which can still be sensitive to *very* heavily corrupted data. A more robust alternative is available, although it is typically much slower.

```
>>> s.decomposition(algorithm="ORNMF", output_dimension=3, method="RobustPGD")
```

9.1.9 Custom decomposition algorithms

HyperSpy supports passing a custom decomposition algorithm, provided it follows the form of a `scikit-learn` estimator. Any object that implements `fit` and `transform` methods is acceptable, including `sklearn.pipeline.Pipeline` and `sklearn.model_selection.GridSearchCV`. You can access the fitted estimator by passing `return_info=True`.

```
>>> # Passing a custom decomposition algorithm
>>> from sklearn.preprocessing import MinMaxScaler
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.decomposition import PCA

>>> pipe = Pipeline([("scaler", MinMaxScaler()), ("PCA", PCA())])
>>> out = s.decomposition(algorithm=pipe, return_info=True)
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=Pipeline(steps=[('scaler', MinMaxScaler()), ('PCA', PCA())])
  output_dimension=None
  centre=None
scikit-learn estimator:
Pipeline(steps=[('scaler', MinMaxScaler()), ('PCA', PCA())])

>>> out
Pipeline(steps=[('scaler', MinMaxScaler()), ('PCA', PCA())])
```

9.2 Blind Source Separation

In some cases it is possible to obtain more physically interpretable set of components using a process called Blind Source Separation (BSS). This largely depends on the particular application. For more information about blind source separation please see [[Hyvarinen2000](#)], and for an example application to EELS analysis, see [[Pena2010](#)].

Warning: The BSS algorithms operate on the result of a previous decomposition analysis. It is therefore necessary to perform a *decomposition* first before calling `blind_source_separation()`, otherwise it will raise an error.

You must provide an integer `number_of_components` argument, or a list of components as the `comp_list` argument. This performs BSS on the chosen number/list of components from the previous decomposition.

To perform blind source separation on the result of a previous decomposition, run the `blind_source_separation()` method, for example:

```
>>> s = hs.signals.Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition(output_dimension=3)
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
```

(continues on next page)

(continued from previous page)

```

output_dimension=3
centre=None

>>> s.blind_source_separation(number_of_components=3)
Blind source separation info:
  number_of_components=3
  algorithm=sklearn_fastica
  diff_order=1
  reverse_component_criterion=factors
  whiten_method=PCA
scikit-learn estimator:
FastICA(tol=1e-10, whiten=False)

# Perform only on the first and third components
>>> s.blind_source_separation(comp_list=[0, 2])
Blind source separation info:
  number_of_components=2
  algorithm=sklearn_fastica
  diff_order=1
  reverse_component_criterion=factors
  whiten_method=PCA
scikit-learn estimator:
FastICA(tol=1e-10, whiten=False)

```

9.2.1 Available algorithms

HyperSpy implements a number of BSS algorithms via the `algorithm` argument. The table below lists the algorithms that are currently available, and includes links to the appropriate documentation for more information on each one.

Table 2: Available blind source separation algorithms in HyperSpy

Algorithm	Method
“sklearn_fastica” (default)	<code>sklearn.decomposition.FastICA</code>
“orthomax”	<code>orthomax()</code>
“FastICA”	<code>mdp.nodes.FastICANode</code>
“JADE”	<code>mdp.nodes.JADENode</code>
“CuBICA”	<code>mdp.nodes.CuBICANode</code>
“TDSEP”	<code>mdp.nodes.TDSEPNode</code>
custom object	An object implementing <code>fit()</code> and <code>transform()</code> methods

Note: Except `orthomax()`, all of the implemented BSS algorithms listed above rely on external packages being available on your system. `sklearn_fastica`, requires `scikit-learn` while `FastICA`, `JADE`, `CuBICA`, `TDSEP` require the `Modular toolkit for Data Processing (MDP)`.

9.2.2 Orthomax

Orthomax rotations are a statistical technique used to clarify and highlight the relationship among factors, by adjusting the coordinates of PCA results. The most common approach is known as “varimax”, which intended to maximize the variance shared among the components while preserving orthogonality. The results of an orthomax rotation following PCA are often “simpler” to interpret than just PCA, since each component has a more discrete contribution to the data.

```
>>> s = hs.signals.Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition(output_dimension=3, print_info=False)

>>> s.blind_source_separation(number_of_components=3, algorithm="orthomax")
Blind source separation info:
  number_of_components=3
  algorithm=orthomax
  diff_order=1
  reverse_component_criterion=factors
  whiten_method=PCA
```

9.2.3 Independent component analysis (ICA)

One of the most common approaches for blind source separation is [Independent Component Analysis \(ICA\)](#). This separates a signal into subcomponents by assuming that the subcomponents are (a) non-Gaussian, and (b) that they are statistically independent from each other.

9.2.4 Custom BSS algorithms

As with *decomposition*, HyperSpy supports passing a custom BSS algorithm, provided it follows the form of a [scikit-learn estimator](#). Any object that implements `fit()` and `transform()` methods is acceptable, including `sklearn.pipeline.Pipeline` and `sklearn.model_selection.GridSearchCV`. You can access the fitted estimator by passing `return_info=True`.

```
>>> # Passing a custom BSS algorithm
>>> from sklearn.preprocessing import MinMaxScaler
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.decomposition import FastICA

>>> pipe = Pipeline([("scaler", MinMaxScaler()), ("ica", FastICA())])
>>> out = s.blind_source_separation(number_of_components=3, algorithm=pipe, return_
↳ info=True, print_info=False)

>>> out
Pipeline(steps=[('scaler', MinMaxScaler()), ('ica', FastICA())])
```

9.3 Cluster analysis

New in version 1.6.

9.3.1 Introduction

Cluster analysis or clustering is the task of grouping a set of measurements such that measurements in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). A HyperSpy signal can represent a number of large arrays of different measurements which can represent spectra, images or sets of parameters. Identifying and extracting trends from large datasets is often difficult and decomposition methods, blind source separation and cluster analysis play an important role in this process.

Cluster analysis, in essence, compares the “distances” (or similar metric) between different sets of measurements and groups those that are closest together. The features it groups can be raw data points, for example, comparing for every navigation dimension all points of a spectrum. However, if the dataset is large, the process of clustering can be computationally intensive so clustering is more commonly used on an extracted set of features or parameters. For example, extraction of two peak positions of interest via a fitting process rather than clustering all spectra points.

In favourable cases, matrix decomposition and related methods can decompose the data into a (ideally small) set of significant loadings and factors. The factors capture a core representation of the features in the data and the loadings provide the mixing ratios of these factors that best describe the original data. Overall, this usually represents a much smaller data volume compared to the original data and can help to identify correlations.

A detailed description of the application of cluster analysis in x-ray spectro-microscopy and further details on the theory and implementation can be found in [Lerotic2004].

9.3.2 Nomenclature

Taking the example of a 1D Signal of dimensions (20, 10|4) containing the dataset, we say there are 200 *samples*. The four measured parameters are the *features*. If we choose to search for 3 clusters within this dataset, we derive three main values:

1. The *labels*, of dimensions (3| 20, 10). Each navigation position is assigned to a cluster. The *labels* of each cluster are boolean arrays that mark the data that has been assigned to the cluster with *True*.
2. The *cluster_distances*, of dimensions (3| 20, 10), which are the distances of all the data points to the centroid of each cluster.
3. The “*cluster signals*”, which are signals that are representative of their clusters. In HyperSpy two are computer: *cluster_sum_signals* and *cluster_centroid_signals*, of dimensions (3| 4), which are the sum of all the cluster signals that belong to each cluster or the signal closest to each cluster centroid respectively.

9.3.3 Clustering functions HyperSpy

All HyperSpy signals have the following methods for clustering analysis:

- `cluster_analysis()`
- `plot_cluster_results()`
- `plot_cluster_labels()`
- `plot_cluster_signals()`
- `plot_cluster_distances()`
- `get_cluster_signals()`

- `get_cluster_labels()`
- `get_cluster_distances()`
- `estimate_number_of_clusters()`
- `plot_cluster_metric()`

The `cluster_analysis()` method can perform cluster analysis using any [scikit-learn clustering](#) algorithms or any other object with a compatible API. This involves importing the relevant algorithm class from scikit-learn.

```
>>> from sklearn.cluster import KMeans
>>> s.cluster_analysis(
...     cluster_source="signal", algorithm=KMeans(n_clusters=3, n_init=8)
...     )
```

For convenience, the default algorithm is the `kmeans` algorithm and is imported internally. All extra keyword arguments are passed to the algorithm when present. Therefore the following code is equivalent to the previous one:

For example:

```
>>> s.cluster_analysis(
...     cluster_source="signal", n_clusters=3, preprocessing="norm", algorithm="kmeans",
...     ←n_init=8
...     )
```

is equivalent to:

`cluster_analysis()` computes the cluster labels. The clusters areas with identical label are averaged to create a set of cluster centres. This averaging can be performed on the `signal` itself, the `BSS` or `decomposition` results or a user supplied signal.

9.3.4 Pre-processing

Cluster analysis measures the distances between features and groups them. It is often necessary to pre-process the features in order to obtain meaningful results.

For example, pre-processing can be useful to reveal clusters when performing cluster analysis of decomposition results. Decomposition methods decompose data into a set of factors and a set of loadings defining the mixing needed to represent the data. If signal 1 is reduced to three components with mixing 0.1 0.5 2.0, and signal 2 is reduced to a mixing of 0.2 1.0 4.0, it should be clear that these represent the same signal but with a scaling difference. Normalization of the data can again be used to remove scaling effects.

Therefore, the pre-processing step will highly influence the results and should be evaluated for the problem under investigation.

All pre-processing methods from (or compatible with) the [scikit-learn pre-processing](#) module can be passed to the `scaling` keyword of the `cluster_analysis()` method. For convenience, the following methods from scikit-learn are available as standard: `standard`, `minmax` and `norm` as standard. Briefly, `norm` treats the features as a vector and normalizes the vector length. `standard` re-scales each feature by removing the mean and scaling to unit variance. `minmax` normalizes each feature between the minimum and maximum range of that feature.

Cluster signals

In HyperSpy *cluster signals* are signals that somehow represent their clusters. The concept is ill-defined, since cluster algorithms only assign data points to clusters. HyperSpy computes 2 cluster signals,

1. `cluster_sum_signals`, which are the sum of all the cluster signals that belong to each cluster.
2. `cluster_centroid_signals`, which is the signal closest to each cluster centroid.

When plotting the “*cluster signals*” we can select any of those above using the `signal` keyword argument:

```
>>> s.plot_cluster_labels(signal="centroid")
```

In addition, it is possible to plot the mean signal over the different clusters:

```
>>> s.plot_cluster_labels(signal="mean")
```

Clustering with user defined algorithms

User developed preprocessing or cluster algorithms can be used in place of the sklearn methods. A preprocessing object needs a `fit_transform` which appropriately scales the data. The example below defines a preprocessing class which normalizes the data then applies a square root to enhances weaker features.

```
>>> class PowerScaling(object):
...     def __init__(self, power=0.5):
...         self.power = power
...     def fit_transform(self, data):
...         norm = np.amax(data, axis=1)
...         scaled_data = data/norm[:, None]
...         scaled_data = scaled_data - np.min(scaled_data)+1.0e-8
...         scaled_data = scaled_data ** self.power
...         return scaled_data
```

The `PowerScaling` class can then be passed to the `cluster_analysis` method for use.

```
>>> ps = PowerScaling()
>>> s.cluster_analysis(
...     cluster_source="decomposition", number_of_components=3, preprocessing=ps
... )
```

For user defined clustering algorithms the class must implement `fit` and have a `label_` attribute that contains the clustering labels. An example template would be:

```
>>> class MyClustering(object):
...     def __init__(self):
...         self.labels_ = None
...     def fit(self, X):
...         self.labels_ = do_something(X)
```

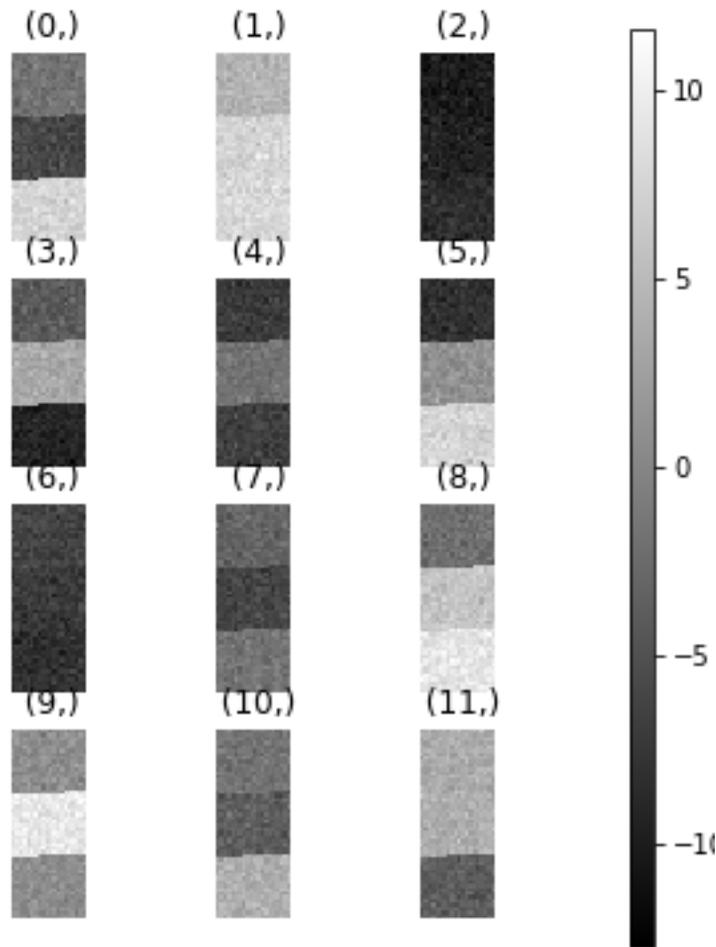
9.3.5 Examples

Clustering using decomposition results

Let's use the `sklearn.datasets.make_blobs()` function supplied by *scikit-learn* to make dummy data to see how clustering might work in practice.

```
>>> from sklearn.datasets import make_blobs
>>> data = make_blobs(
...     n_samples=1000,
...     n_features=100,
...     centers=3,
...     shuffle=False,
...     random_state=1)[0].reshape(50, 20, 100)
>>> s = hs.signals.Signal1D(data)
```

```
>>> hs.plot.plot_images(s.T.inav[:9], axes_decor="off")
```

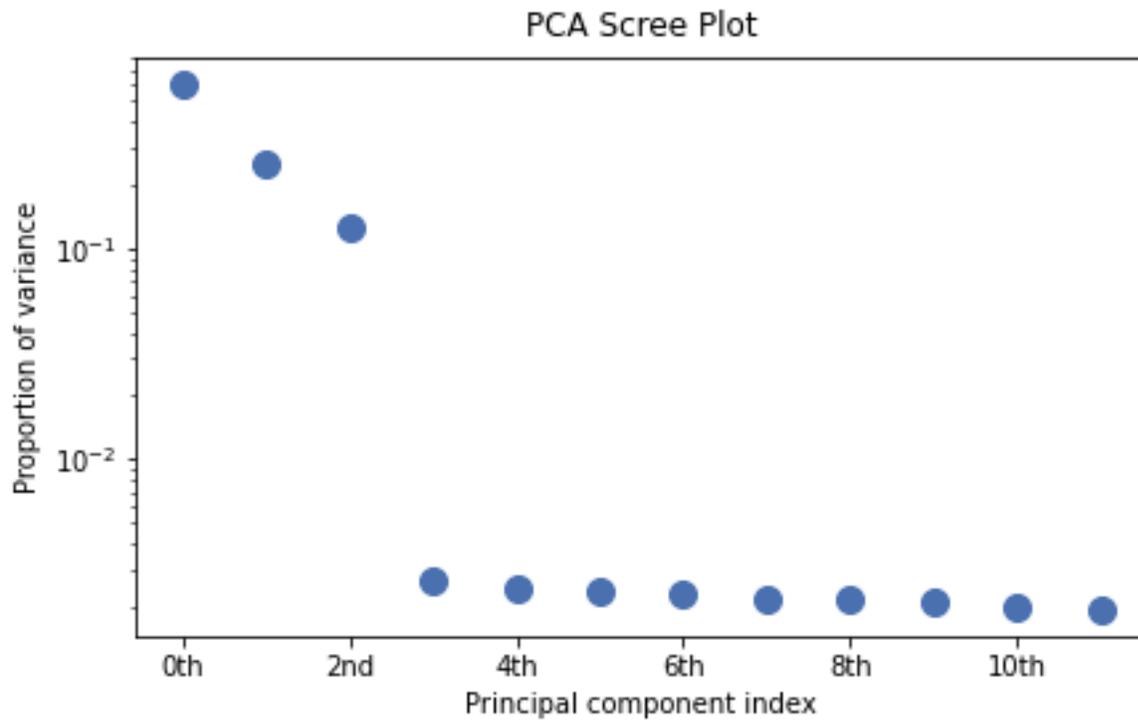


To see how cluster analysis works it's best to first examine the signal. Moving around the image you should be able to see 3 distinct regions in which the 1D signal modulates slightly.

```
>>> s.plot()
```

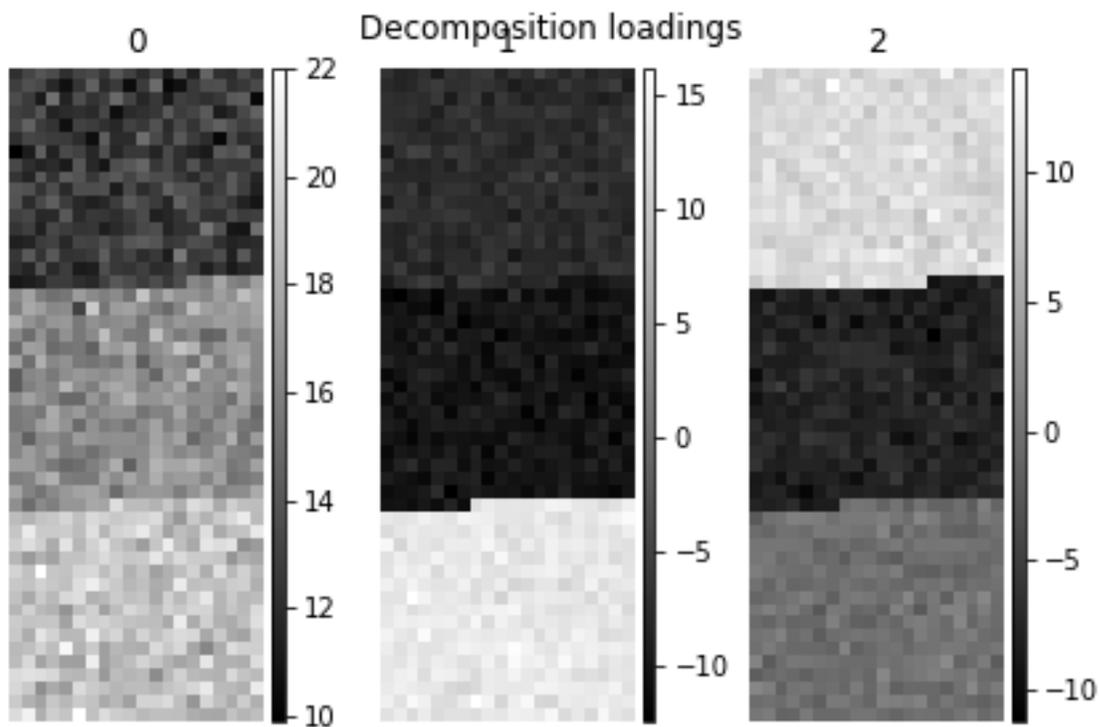
Let's perform SVD to reduce the dimensionality of the dataset by exploiting redundancies:

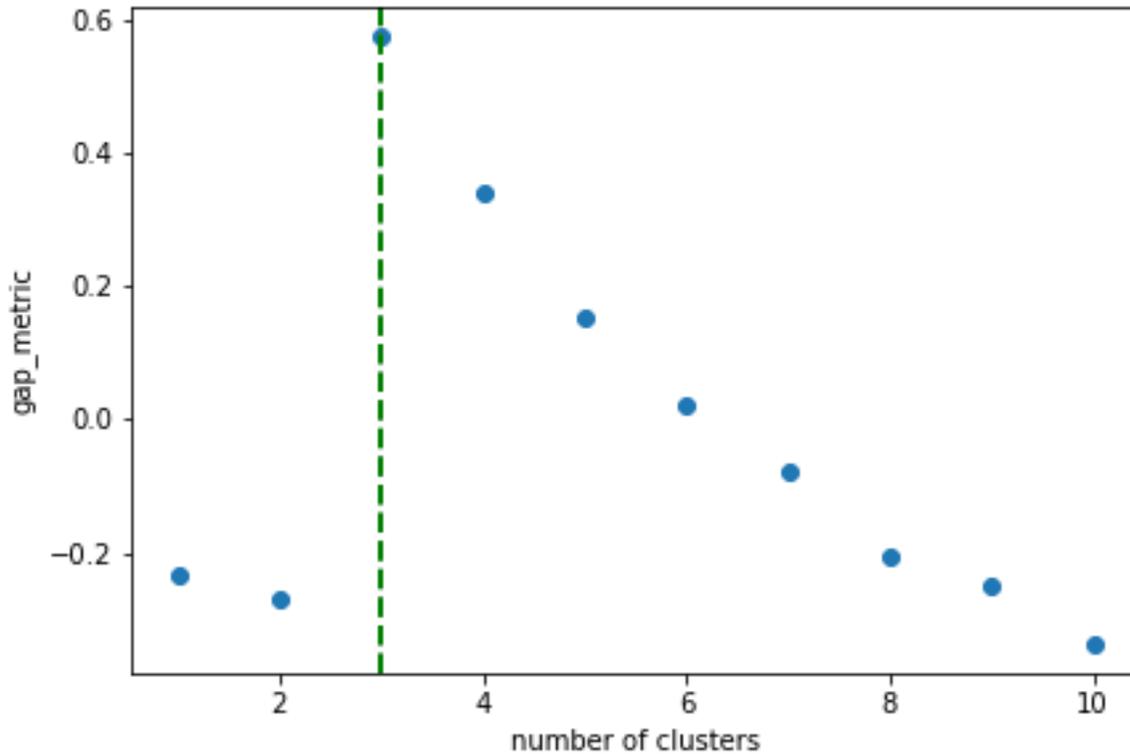
```
>>> s.decomposition()
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=None
  centre=None
>>> s.plot_explained_variance_ratio()
<Axes: title={'center': '\nPCA Scree Plot'}, xlabel='Principal component index', ylabel=
↪ 'Proportion of variance'>
```



From the scree plot we deduce that, as expected, that the dataset can be reduce to 3 components. Let's plot their loadings:

```
>>> s.plot_decomposition_loadings(comp_ids=3, axes_decor="off")
```





The optimal number of clusters can be set or accessed from the learning results

```
>>> s.learning_results.number_of_clusters
3
```

Clustering using another signal as source

In this example we will perform clustering analysis on the position of two peaks. The signals containing the position of the peaks can be computed for example using *curve fitting*. Given an existing fitted model, the parameters can be extracted as signals and stacked. Clustering can then be applied as described previously to identify trends in the fitted results.

Let's start by creating a suitable synthetic dataset.

```
>>> import hyperspy.api as hs
>>> import numpy as np
>>> s_dummy = hs.signals.Signal1D(np.zeros((64, 64, 1000)))
>>> s_dummy.axes_manager.signal_axes[0].scale = 2e-3
>>> s_dummy.axes_manager.signal_axes[0].units = "eV"
>>> s_dummy.axes_manager.signal_axes[0].name = "energy"
>>> m = s_dummy.create_model()
>>> m.append(hs.model.components1D.GaussianHF(fwhm=0.2))
>>> m.append(hs.model.components1D.GaussianHF(fwhm=0.3))
>>> m.components.GaussianHF.centre.map["values"][:32, :] = .3 + .1
>>> m.components.GaussianHF.centre.map["values"][32:, :] = .7 + .1
>>> m.components.GaussianHF_0.centre.map["values"][:, 32:] = m.components.GaussianHF.
↵centre.map["values"][:, 32:] * 2
>>> m.components.GaussianHF_0.centre.map["values"][:, :32] = m.components.GaussianHF.
```

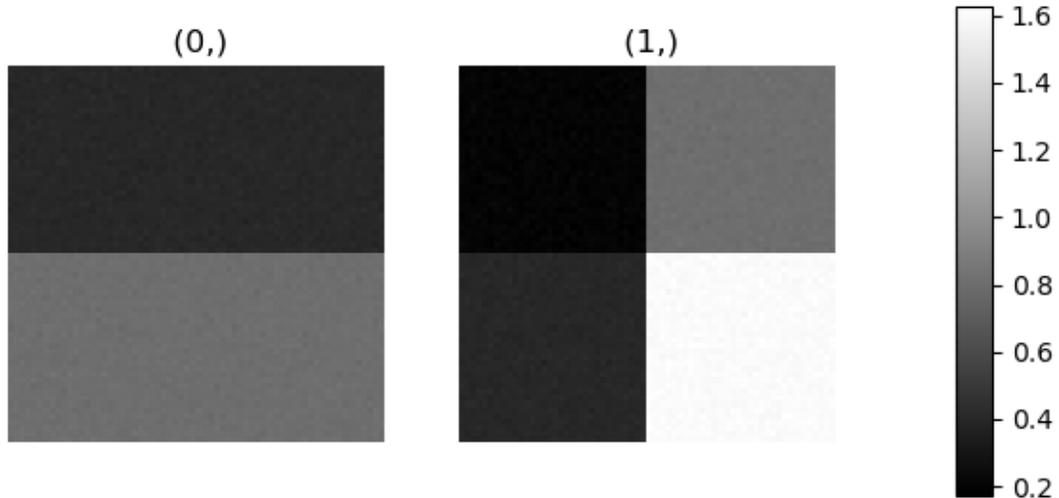
(continues on next page)

(continued from previous page)

```

↪ centre.map["values"][:, :32] * 0.5
>>> for component in m:
...     component.centre.map["is_set"][:] = True
...     component.centre.map["values"][:] += np.random.normal(size=(64, 64)) * 0.01
>>> s = m.as_signal()
>>> stack = [component.centre.as_signal() for component in m]
>>> hs.plot.plot_images(stack, axes_decor="off", colorbar="single", subtitle="")

```



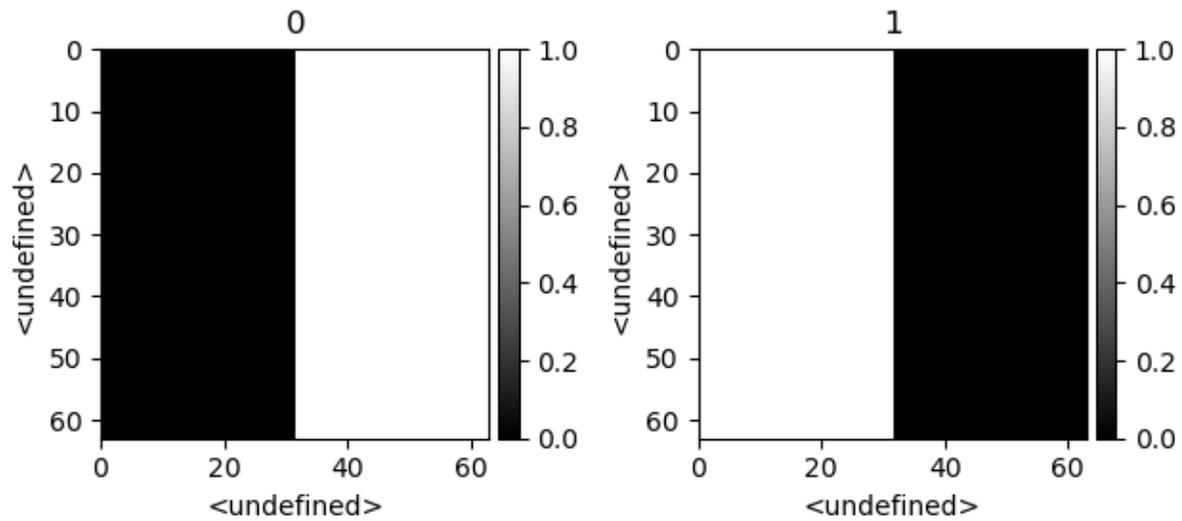
Let's now perform cluster analysis on the stack and calculate the centres using the spectrum image. Notice that we don't need to fit the model to the data because this is a synthetic dataset. When analysing experimental data you will need to fit the model first. Also notice that here we need to pre-process the dataset by normalization in order to reveal the clusters due to the proportionality relationship between the position of the peaks.

```

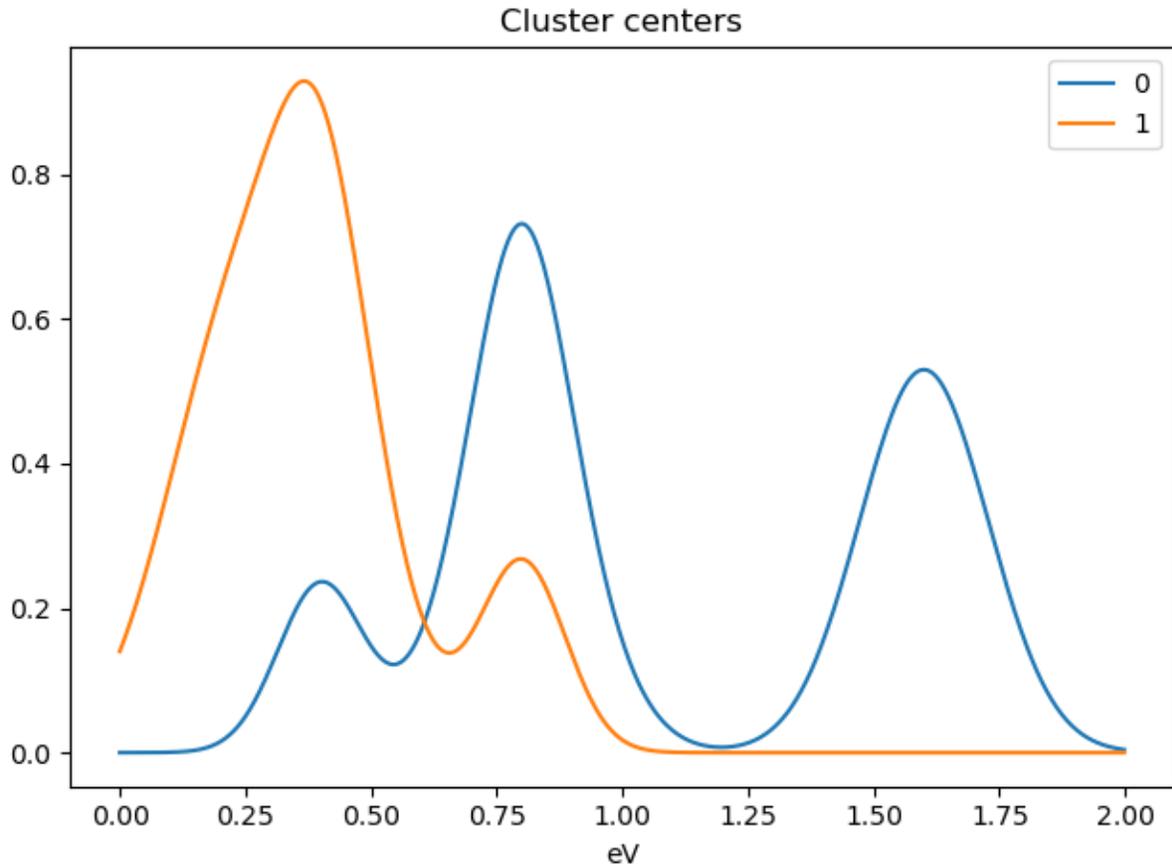
>>> stack = hs.stack([component.centre.as_signal() for component in m])
>>> s.estimate_number_of_clusters(cluster_source=stack.T, preprocessing="norm")
2
>>> s.cluster_analysis(cluster_source=stack.T, source_for_centers=s, n_clusters=2,
↪ preprocessing="norm")
>>> s.plot_cluster_labels()

```

Cluster labels of from fitted model

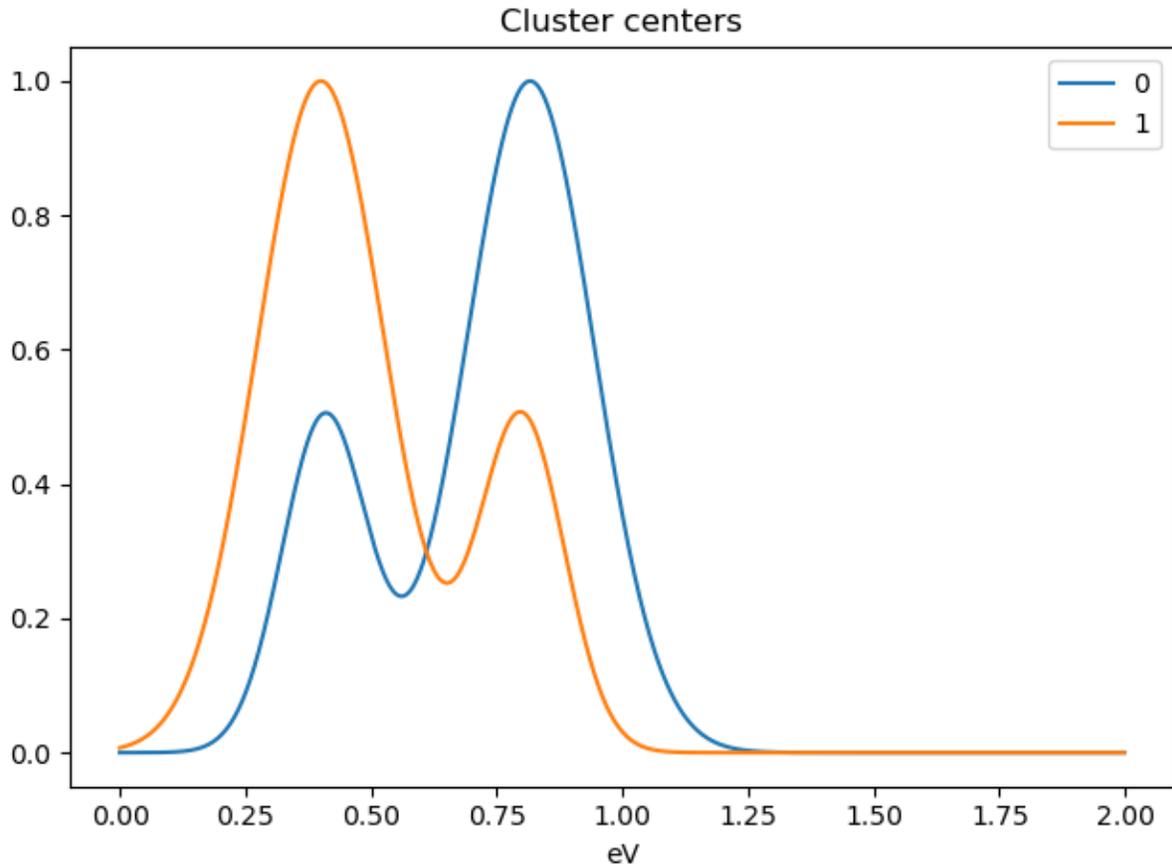


```
>>> s.plot_cluster_signals(signal="mean")
```



Notice that in this case averaging or summing the signals of each cluster is not appropriate, since the clustering criterium is the ratio between the peaks positions. A better alternative is to plot the signals closest to the centroids:

```
>>> s.plot_cluster_signals(signal="centroid")
```



9.4 Visualizing results

HyperSpy includes a number of plotting methods for visualizing the results of decomposition and blind source separation analyses. All the methods begin with `plot_`.

9.4.1 Scree plots

Note: Scree plots are only available for the "SVD" and "PCA" algorithms.

PCA will sort the components in the dataset in order of decreasing variance. It is often useful to estimate the dimensionality of the data by plotting the explained variance against the component index. This plot is sometimes called a scree plot. For most datasets, the values in a scree plot will decay rapidly, eventually becoming a slowly descending line.

To obtain a scree plot for your dataset, run the `plot_explained_variance_ratio()` method:

```
>>> s.plot_explained_variance_ratio(n=20)
```

The point at which the scree plot becomes linear (often referred to as the “elbow”) is generally judged to be a good estimation of the dimensionality of the data (or equivalently, the number of components that should be retained - see

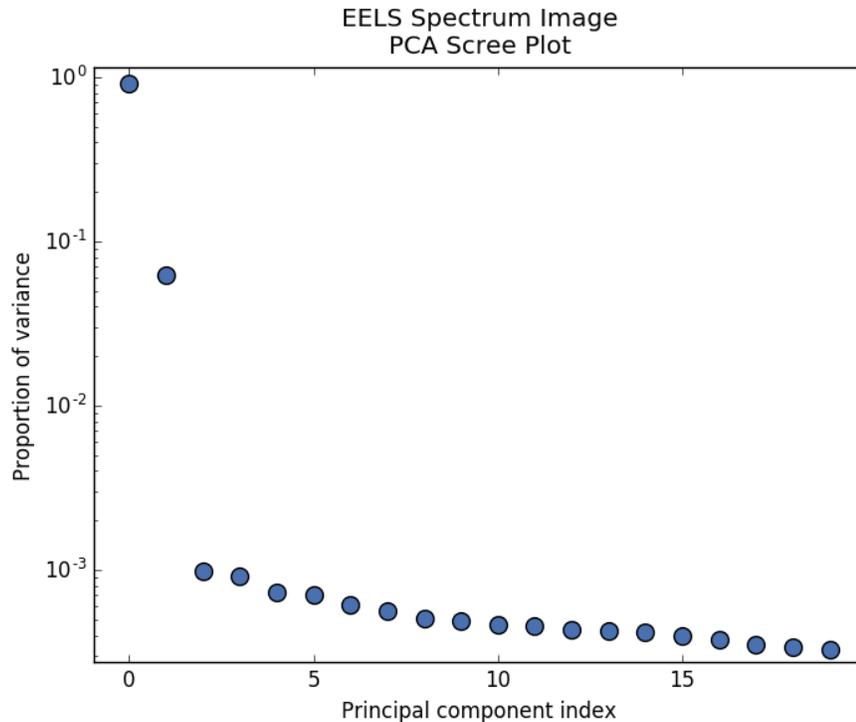


Fig. 2: PCA scree plot

below). Components to the left of the elbow are considered part of the “signal”, while components to the right are considered to be “noise”, and thus do not explain any significant features of the data.

By specifying a `threshold` value, a cutoff line will be drawn at the total variance specified, and the components above this value will be styled distinctly from the remaining components to show which are considered signal, as opposed to noise. Alternatively, by providing an integer value for `threshold`, the line will be drawn at the specified component (see below).

Note that in the above scree plot, the first component has index 0. This is because Python uses zero-based indexing. To switch to a “number-based” (rather than “index-based”) notation, specify the `xaxis_type` parameter:

```
>>> s.plot_explained_variance_ratio(n=20, threshold=4, xaxis_type='number')
```

The number of significant components can be estimated and a vertical line drawn to represent this by specifying `vline=True`. In this case, the “elbow” is found in the variance plot by estimating the distance from each point in the variance plot to a line joining the first and last points of the plot, and then selecting the point where this distance is largest.

If multiple maxima are found, the index corresponding to the first occurrence is returned. As the index of the first component is zero, the number of significant PCA components is the elbow index position + 1. More details about the elbow-finding technique can be found in [Satopää2011], and in the documentation for `estimate_elbow_position()`.

These options (together with many others), can be customized to develop a figure of your liking. See the documentation of `plot_explained_variance_ratio()` for more details.

Sometimes it can be useful to get the explained variance ratio as a spectrum. For example, to plot several scree plots obtained with different data pre-treatments in the same figure, you can combine `plot_spectra()` with `get_explained_variance_ratio()`.

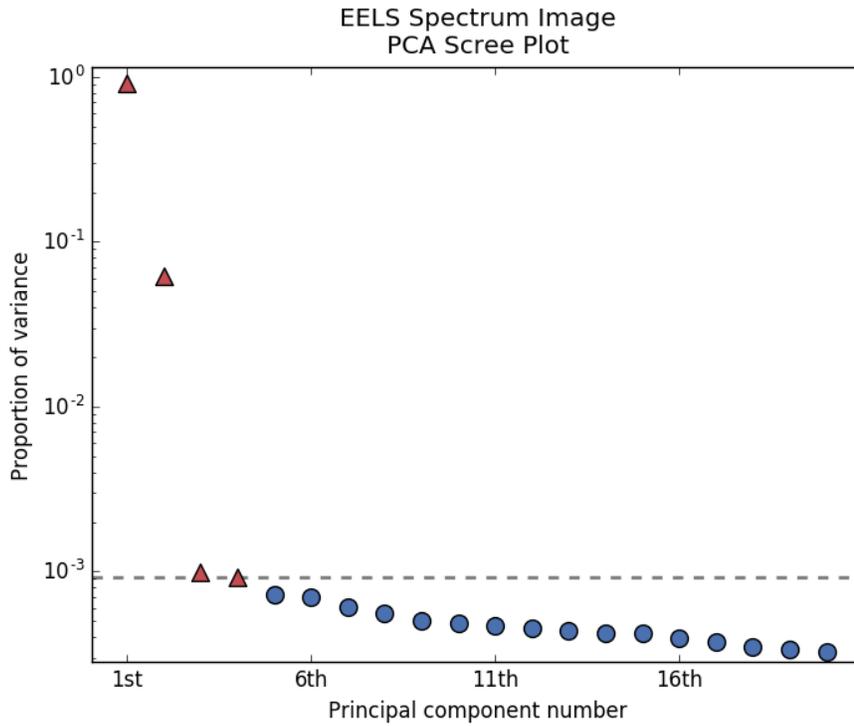
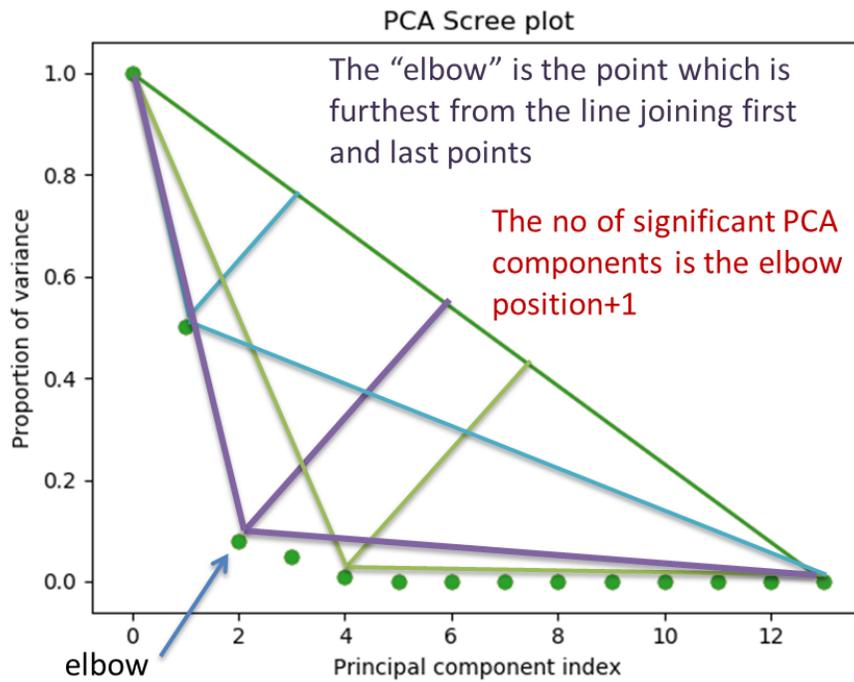


Fig. 3: PCA scree plot with number-based axis labeling and a threshold value specified



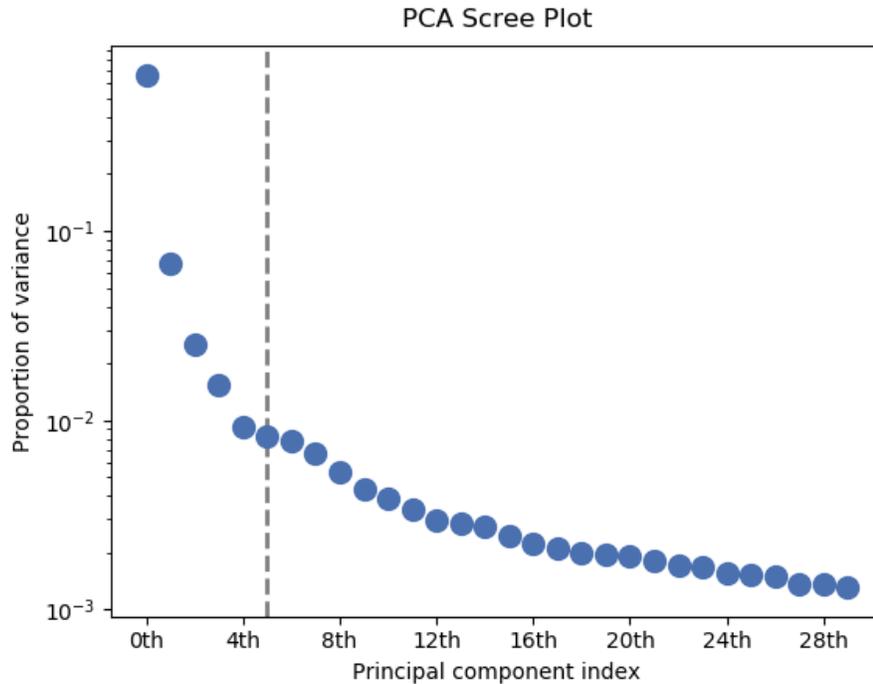


Fig. 4: PCA scree plot with number-based axis labeling and an estimate of the no of significant positions based on the “elbow” position

9.4.2 Decomposition plots

HyperSpy provides a number of methods for visualizing the factors and loadings found by a decomposition analysis. To plot everything in a compact form, use `plot_decomposition_results()`.

You can also plot the factors and loadings separately using the following methods. It is recommended that you provide the number of factors or loadings you wish to visualise, since the default is to plot all of them.

- `plot_decomposition_factors()`
- `plot_decomposition_loadings()`

9.4.3 Blind source separation plots

Visualizing blind source separation results is much the same as decomposition. You can use `plot_bss_results()` for a compact display, or instead:

- `plot_bss_factors()`
- `plot_bss_loadings()`

9.4.4 Clustering plots

Visualizing cluster results is much the same as decomposition. You can use `plot_bss_results()` for a compact display, or instead:

- `plot_cluster_results()`.
- `plot_cluster_signals()`.
- `plot_cluster_labels()`.

9.5 Export results

9.5.1 Obtain the results as BaseSignal instances

The decomposition and BSS results are internally stored as numpy arrays in the `BaseSignal` class. Frequently it is useful to obtain the decomposition/BSS factors and loadings as HyperSpy signals, and HyperSpy provides the following methods for that purpose:

- `get_decomposition_loadings()`
- `get_decomposition_factors()`
- `get_bss_loadings()`
- `get_bss_factors()`

9.5.2 Save and load results

Save in the main file

If you save the dataset on which you've performed machine learning analysis in the `HSpy-HDF5` format (the default in HyperSpy, see [Saving](#)), the result of the analysis is also saved in the same file automatically, and it is loaded along with the rest of the data when you next open the file.

Note: This approach currently supports storing one decomposition and one BSS result, which may not be enough for your purposes.

Save to an external file

Alternatively, you can save the results of the current machine learning analysis to a separate file with the `save()` method:

```
>>> # Save the result of the analysis
>>> s.learning_results.save('my_results.npz')

>>> # Load back the results
>>> s.learning_results.load('my_results.npz')
```

Export in different formats

You can also export the results of a machine learning analysis to any format supported by HyperSpy with the following methods:

- `export_decomposition_results()`
- `export_bss_results()`

These methods accept many arguments to customise the way in which the data is exported, so please consult the method documentation. The options include the choice of file format, the prefixes for loadings and factors, saving figures instead of data and more.

Warning: Data exported in this way cannot be easily loaded into HyperSpy's machine learning structure.

MODEL FITTING

HyperSpy can perform curve fitting of one-dimensional signals (spectra) and two-dimensional signals (images) in n -dimensional data sets. Models are defined by adding individual functions (components in HyperSpy's terminology) to a *BaseModel* instance. Those individual components are then summed to create the final model function that can be fitted to the data, by adjusting the free parameters of the individual components.

Models can be created and fit to experimental data in both one and two dimensions i.e. spectra and images respectively. Most of the syntax is identical in either case. A one-dimensional model is created when a model is created for a *Signal1D* whereas a two-dimensional model is created for a *Signal2D*.

Note: Plotting and analytical gradient-based fitting methods are not yet implemented for the *Model2D* class.

10.1 Creating a model

A *Model1D* can be created for data in the *Signal1D* class using the `create_model()` method:

```
>>> s = hs.signals.Signal1D(np.arange(300).reshape(30, 10))
>>> m = s.create_model() # Creates the 1D-Model and assign it to m
```

Similarly, a *Model2D* can be created for data in the *Signal2D* class using the `create_model()` method:

```
>>> im = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> mod = im.create_model() # Create the 2D-Model and assign it to mod
```

The syntax for creating both one-dimensional and two-dimensional models is thus identical for the user in practice. When a model is created you may be prompted to provide important information not already included in the datafile, e.g. if `s` is EELS data, you may be asked for the accelerating voltage, convergence and collection semi-angles etc.

Note:

- Before creating a model verify that the `is_binned` attribute of the signal axis is set to the correct value because the resulting model depends on this parameter. See *Binned and unbinned signals* for more details.
 - When importing data that has been binned using other software, in particular Gatan's DM, the stored values may be the averages of the binned channels or pixels, instead of their sum, as would be required for proper statistical analysis. We therefore cannot guarantee that the statistics will be valid, and so strongly recommend that all pre-fitting binning is performed using Hyperspy.
-

10.2 Model components

In HyperSpy a model consists of a sum of individual components. For convenience, HyperSpy provides a number of pre-defined model components as well as mechanisms to create your own components.

10.2.1 Pre-defined model components

Various components are available in one (*components1D*) and two-dimensions (*components2D*) to construct a model.

The following general components are currently available for one-dimensional models:

- *Arctan*
- *Bleasdale*
- *Doniach*
- *Erf*
- *Exponential*
- *Expression*
- *Gaussian*
- *GaussianHF*
- *HeavisideStep*
- *Logistic*
- *Lorentzian*
- *Offset*
- *Polynomial*
- *PowerLaw*
- *ScalableFixedPattern*
- *SkewNormal*
- *Voigt*
- *SplitVoigt*

The following components are currently available for two-dimensional models:

- *Expression*
- *Gaussian2D*

However, this doesn't mean that you have to limit yourself to this meagre list of functions. As discussed below, it is very easy to turn a mathematical, fixed-pattern or Python function into a component.

10.2.2 Define components from a mathematical expression

The easiest way to turn a mathematical expression into a component is using the *Expression* component. For example, the following is all you need to create a *Gaussian* component with more sensible parameters for spectroscopy than the one that ships with HyperSpy:

```
>>> g = hs.model.components1D.Expression(
... expression="height * exp(-(x - x0) ** 2 * 4 * log(2)/ fwhm ** 2)",
... name="Gaussian",
... position="x0",
... height=1,
... fwhm=1,
... x0=0,
... module="numpy")
```

If the expression is inconvenient to write out in full (e.g. it's long and/or complicated), multiple substitutions can be given, separated by semicolons. Both symbolic and numerical substitutions are allowed:

```
>>> expression = "h / sqrt(p2) ; p2 = 2 * m0 * e1 * x * brackets;"
>>> expression += "brackets = 1 + (e1 * x) / (2 * m0 * c * c) ;"
>>> expression += "m0 = 9.1e-31 ; c = 3e8; e1 = 1.6e-19 ; h = 6.6e-34"
>>> wavelength = hs.model.components1D.Expression(
... expression=expression,
... name="Electron wavelength with voltage")
```

Expression uses *Sympy* internally to turn the string into a function. By default it “translates” the expression using *numpy*, but often it is possible to boost performance by using *numexpr* instead.

It can also create 2D components with optional rotation. In the following example we create a 2D Gaussian that rotates around its center:

```
>>> g = hs.model.components2D.Expression(
... "k * exp(-((x-x0)**2 / (2 * sx ** 2) + (y-y0)**2 / (2 * sy ** 2)))",
... "Gaussian2d", add_rotation=True, position=("x0", "y0"),
... module="numpy", )
```

10.2.3 Define new components from a Python function

Of course *Expression* is only useful for analytical functions. You can define more general components modifying the following template to suit your needs:

```
from hyperspy.component import Component

class MyComponent(Component):

    """
    """

    def __init__(self, parameter_1=1, parameter_2=2):
        # Define the parameters
        Component.__init__(self, ('parameter_1', 'parameter_2'))

        # Optionally we can set the initial values
```

(continues on next page)

(continued from previous page)

```
self.parameter_1.value = parameter_1
self.parameter_2.value = parameter_2

# The units (optional)
self.parameter_1.units = 'Tesla'
self.parameter_2.units = 'Kociak'

# Once defined we can give default values to the attribute
# For example we fix the attribute_1 (optional)
self.parameter_1.attribute_1.free = False

# And we set the boundaries (optional)
self.parameter_1.bmin = 0.
self.parameter_1.bmax = None

# Optionally, to boost the optimization speed we can also define
# the gradients of the function we the syntax:
# self.parameter.grad = function
self.parameter_1.grad = self.grad_parameter_1
self.parameter_2.grad = self.grad_parameter_2

# Define the function as a function of the already defined parameters,
# x being the independent variable value
def function(self, x):
    p1 = self.parameter_1.value
    p2 = self.parameter_2.value
    return p1 + x * p2

# Optionally define the gradients of each parameter
def grad_parameter_1(self, x):
    """
    Returns d(function)/d(parameter_1)
    """
    return 0

def grad_parameter_2(self, x):
    """
    Returns d(function)/d(parameter_2)
    """
    return x
```

10.2.4 Define components from a fixed-pattern

The *ScalableFixedPattern* component enables fitting a pattern (in the form of a *Signal1D* instance) to data by shifting (*shift*) and scaling it in the x and y directions using the *xscale* and *yscale* parameters respectively.

10.3 Adding components to the model

To print the current components in a model use *components*. A table with component number, attribute name, component name and component type will be printed:

```
>>> s = hs.signals.Signal1D(np.arange(100))
>>> m = s.create_model()
>>> m
<Model1D>
>>> m.components
```

#	Attribute Name	Component Name	Component Type
----	-----	-----	-----

Note: Sometimes components may be created automatically. For example, if the *Signal1D* is recognised as EELS data, a power-law background component may automatically be added to the model. Therefore, the table above may not all may empty on model creation.

To add a component to the model, first we have to create an instance of the component. Once the instance has been created we can add the component to the model using the *append()* and *extend()* methods for one or more components respectively.

As an example, let's add several *Gaussian* components to the model:

```
>>> gaussian = hs.model.components1D.Gaussian() # Create a Gaussian comp.
>>> m.append(gaussian) # Add it to the model
>>> m.components # Print the model components
```

#	Attribute Name	Component Name	Component Type
----	-----	-----	-----
0	Gaussian	Gaussian	Gaussian

```
>>> gaussian2 = hs.model.components1D.Gaussian() # Create another gaussian
>>> gaussian3 = hs.model.components1D.Gaussian() # Create a third gaussian
```

We could use the *append()* method twice to add the two Gaussians, but when adding multiple components it is handier to use the *extend()* method that enables adding a list of components at once.

```
>>> m.extend((gaussian2, gaussian3)) # note the double parentheses!
>>> m.components
```

#	Attribute Name	Component Name	Component Type
----	-----	-----	-----
0	Gaussian	Gaussian	Gaussian
1	Gaussian_0	Gaussian_0	Gaussian
2	Gaussian_1	Gaussian_1	Gaussian

We can customise the name of the components.

```
>>> gaussian.name = 'Carbon'
>>> gaussian2.name = 'Long Hydrogen name'
>>> gaussian3.name = 'Nitrogen'
>>> m.components
```

#	Attribute Name	Component Name	Component Type
0	Carbon	Carbon	Gaussian
1	Long_Hydrogen_name	Long Hydrogen name	Gaussian
2	Nitrogen	Nitrogen	Gaussian

Notice that two components cannot have the same name:

```
>>> gaussian2.name = 'Carbon'
Traceback (most recent call last):
  File "<ipython-input-5-2b5669fae54a>", line 1, in <module>
    g2.name = "Carbon"
  File "/home/fjd29/Python/hyperspy/hyperspy/component.py", line 466, in
    name "the name " + str(value))
ValueError: Another component already has the name Carbon
```

It is possible to access the components in the model by their name or by the index in the model.

```
>>> m.components
```

#	Attribute Name	Component Name	Component Type
0	Carbon	Carbon	Gaussian
1	Long_Hydrogen_name	Long Hydrogen name	Gaussian
2	Nitrogen	Nitrogen	Gaussian

```
>>> m[0]
<Carbon (Gaussian component)>
>>> m["Long Hydrogen name"]
<Long Hydrogen name (Gaussian component)>
```

In addition, the components can be accessed in the `components` Model attribute. This is specially useful when working in interactive data analysis with IPython because it enables tab completion.

```
>>> m.components
```

#	Attribute Name	Component Name	Component Type
0	Carbon	Carbon	Gaussian
1	Long_Hydrogen_name	Long Hydrogen name	Gaussian
2	Nitrogen	Nitrogen	Gaussian

```
>>> m.components.Long_Hydrogen_name
<Long Hydrogen name (Gaussian component)>
```

It is possible to “switch off” a component by setting its active attribute to False. When a component is switched off, to all effects it is as if it was not part of the model. To switch it back on simply set the active attribute back to True.

In multi-dimensional signals it is possible to store the value of the active attribute at each navigation index. To enable this feature for a given component set the `active_is_multidimensional` attribute to `True`.

```
>>> s = hs.signals.Signal1D(np.arange(100).reshape(10,10))
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
```

(continues on next page)

(continued from previous page)

```
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1,g2])
>>> g1.active_is_multidimensional = True
>>> m.set_component_active_value(False)
>>> m.set_component_active_value(True, only_current=True)
```

10.4 Indexing the model

Often it is useful to consider only part of the model - for example at a particular location (i.e. a slice in the navigation space) or energy range (i.e. a slice in the signal space). This can be done using exactly the same syntax that we use for signal *indexing*. *red_chisq* and *dof* are automatically recomputed for the resulting slices.

```
>>> s = hs.signals.Signal1D(np.arange(100).reshape(10,10))
>>> m = s.create_model()
>>> m.append(hs.model.components1D.Gaussian())
>>> # select first three navigation pixels and last five signal channels
>>> m1 = m.inav[:3].isig[-5:]
>>> m1.signal
<Signal1D, title: , dimensions: (3|5)>
```

10.5 Getting and setting parameter values and attributes

10.5.1 Getting parameter values

print_current_values() prints the properties of the parameters of the components in the current coordinates. In the Jupyter Notebook, the default view is HTML-formatted, which allows for formatted copying into other software, such as Excel. One can also filter for only active components and only showing component with free parameters with the arguments *only_active* and *only_free*, respectively.

The current values of a particular component can be printed using the *print_current_values()* method.

```
>>> s = exspy.data.EDS_SEM_TM002()
>>> m = s.create_model()
>>> m.fit()
>>> G = m[1]
>>> G.print_current_values()
Gaussian: Al_Ka
Active: True
Parameter Name | Free | Value | Std | Min
===== | ===== | ===== | ===== | =====
A | True | 62894.6824 | 1039.40944 | 0.0
sigma | False | 0.03253440 | None | None
centre | False | 1.4865 | None | None
```

The current coordinates can be either set by navigating the *plot()*, or specified by pixel indices in *m.axes_manager.indices* or as calibrated coordinates in *m.axes_manager.coordinates*.

parameters contains a list of the parameters of a component and *free_parameters* lists only the free parameters.

The value of a particular parameter in the current coordinates can be accessed by `component.Parameter.value` (e.g. `Gaussian.A.value`). To access an array of the value of the parameter across all navigation pixels, `component.Parameter.map['values']` (e.g. `Gaussian.A.map["values"]`) can be used. On its own, `component.Parameter.map` returns a NumPy array with three elements: 'values', 'std' and 'is_set'. The first two give the value and standard error for each index. The last element shows whether the value has been set in a given index, either by a fitting procedure or manually.

If a model contains several components with the same parameters, it is possible to change them all by using `set_parameters_value()`:

```
>>> s = hs.signals.Signal1D(np.arange(100).reshape(10,10))
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1,g2])
>>> m.set_parameters_value('A', 20)
>>> g1.A.map['values']
array([20., 20., 20., 20., 20., 20., 20., 20., 20., 20.])
>>> g2.A.map['values']
array([20., 20., 20., 20., 20., 20., 20., 20., 20., 20.])
>>> m.set_parameters_value('A', 40, only_current=True)
>>> g1.A.map['values']
array([40., 20., 20., 20., 20., 20., 20., 20., 20., 20.])
>>> m.set_parameters_value('A',30, component_list=[g2])
>>> g2.A.map['values']
array([30., 30., 30., 30., 30., 30., 30., 30., 30., 30.])
>>> g1.A.map['values']
array([40., 20., 20., 20., 20., 20., 20., 20., 20., 20.])
```

10.5.2 Setting Parameters free / not free

To set the free state of a parameter change the `free` attribute. To change the free state of all parameters in a component to `True` use `set_parameters_free()`, and `set_parameters_not_free()` for setting them to `False`. Specific parameter-names can also be specified by using `parameter_name_list`, shown in the example:

```
>>> g = hs.model.components1D.Gaussian()
>>> g.free_parameters
(<Parameter A of Gaussian component>, <Parameter centre of Gaussian component>,
 →<Parameter sigma of Gaussian component>)
>>> g.set_parameters_not_free()
>>> g.set_parameters_free(parameter_name_list=['A','centre'])
>>> g.free_parameters
(<Parameter A of Gaussian component>, <Parameter centre of Gaussian component>)
```

Similar functions exist for `BaseModel`: `set_parameters_free()` and `set_parameters_not_free()`. Which sets the free states for the parameters in components in a model. Specific components and parameter-names can also be specified. For example:

```
>>> g1 = hs.model.components1D.Gaussian()
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1,g2])
>>> m.set_parameters_not_free()
>>> g1.free_parameters
```

(continues on next page)

(continued from previous page)

```

()
>>> g2.free_parameters
()
>>> m.set_parameters_free(parameter_name_list=['A'])
>>> g1.free_parameters
(<Parameter A of Gaussian_1 component>,)
>>> g2.free_parameters
(<Parameter A of Gaussian_2 component>,)
>>> m.set_parameters_free([g1], parameter_name_list=['sigma'])
>>> g1.free_parameters
(<Parameter A of Gaussian_1 component>, <Parameter sigma of Gaussian_1 component>)
>>> g2.free_parameters
(<Parameter A of Gaussian_2 component>,)

```

10.5.3 Setting twin parameters

The value of a parameter can be coupled to the value of another by setting the *twin* attribute:

```

>>> s = hs.signals.Signal1D(np.arange(100))
>>> m = s.create_model()

>>> gaussian = hs.model.components1D.Gaussian()
>>> gaussian2 = hs.model.components1D.Gaussian() # Create another gaussian
>>> gaussian3 = hs.model.components1D.Gaussian() # Create a third gaussian
>>> gaussian.name = 'Carbon'
>>> gaussian2.name = 'Long Hydrogen name'
>>> gaussian3.name = 'Nitrogen'
>>> m.extend((gaussian, gaussian2, gaussian3))

>>> gaussian.parameters # Print the parameters of the Gaussian components
(<Parameter A of Carbon component>, <Parameter centre of Carbon component>, <Parameter_
↳sigma of Carbon component>)
>>> gaussian.centre.free = False # Fix the centre
>>> gaussian.free_parameters # Print the free parameters
(<Parameter A of Carbon component>, <Parameter sigma of Carbon component>)
>>> m.print_current_values(only_free=True) # Print the values of all free parameters.
Model1D:
CurrentComponentValues: Carbon
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
A | True | 1.0 | None | 0.0 | None | True
sigma | True | 1.0 | None | 0.0 | None | False

CurrentComponentValues: Long Hydrogen name
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
A | True | 1.0 | None | 0.0 | None | True
centre | True | 0.0 | None | None | None | False
sigma | True | 1.0 | None | 0.0 | None | False

```

(continues on next page)

(continued from previous page)

```

CurrentComponentValues: Nitrogen
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
          A | True | 1.0 | None | 0.0 | None | True
       centre | True | 0.0 | None | None | None | False
          sigma | True | 1.0 | None | 0.0 | None | False

>>> # Couple the A parameter of gaussian2 to the A parameter of gaussian 3:
>>> gaussian2.A.twin = gaussian3.A
>>> gaussian2.A.value = 10 # Set the gaussian2 A value to 10
>>> gaussian3.print_current_values()
CurrentComponentValues: Nitrogen
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
          A | True | 10.0 | None | 0.0 | None | True
       centre | True | 0.0 | None | None | None | False
          sigma | True | 1.0 | None | 0.0 | None | False

>>> gaussian3.A.value = 5 # Set the gaussian1 centre value to 5
>>> gaussian2.print_current_values()
CurrentComponentValues: Long Hydrogen name
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
          A | Twinned | 5.0 | None | 0.0 | None | True
       centre | True | 0.0 | None | None | None | False
          sigma | True | 1.0 | None | 0.0 | None | False

```

Deprecated since version 1.2.0: Setting the `twin_function` and `twin_inverse_function` attributes, set the `twin_function_expr` and `twin_inverse_function_expr` attributes instead.

New in version 1.2.0: `twin_function_expr` and `twin_inverse_function_expr`.

By default the coupling function is the identity function. However it is possible to set a different coupling function by setting the `twin_function_expr` and `twin_inverse_function_expr` attributes. For example:

```

>>> gaussian2.A.twin_function_expr = "x**2"
>>> gaussian2.A.twin_inverse_function_expr = "sqrt(abs(x))"
>>> gaussian2.A.value = 4
>>> gaussian3.print_current_values()
CurrentComponentValues: Nitrogen
Active: True
Parameter Name | Free | Value | Std | Min | Max | Linear
===== | ===== | ===== | ===== | ===== | ===== | =====
          A | True | 2.0 | None | 0.0 | None | True
       centre | True | 0.0 | None | None | None | False
          sigma | True | 1.0 | None | 0.0 | None | False

>>> gaussian3.A.value = 4
>>> gaussian2.print_current_values()
CurrentComponentValues: Long Hydrogen name

```

(continues on next page)

(continued from previous page)

Parameter Name	Free	Value	Std	Min	Max	Linear
A	Twinned	16.0	None	0.0	None	True
centre	True	0.0	None	None	None	False
sigma	True	1.0	None	0.0	None	False

10.5.4 Batch setting of parameter attributes

The following model methods can be used to ease the task of setting some important parameter attributes. These can also be used on a per-component basis, by calling them on individual components.

- `set_parameters_not_free()`
- `set_parameters_free()`
- `set_parameters_value()`

10.6 Fitting the model to the data

To fit the model to the data at the current coordinates (e.g. to fit one spectrum at a particular point in a spectrum-image), use `fit()`. HyperSpy implements a number of different optimization approaches, each of which can have particular benefits and/or drawbacks depending on your specific application. A good approach to choosing an optimization approach is to ask yourself the question “Do you want to...”:

- Apply bounds to your model parameter values?
- Use gradient-based fitting algorithms to accelerate your fit?
- Estimate the standard deviations of the parameter values found by the fit?
- Fit your data in the least-squares sense, or use another loss function?
- Find the global optima for your parameters, or is a local optima acceptable?

10.6.1 Optimization algorithms

The following table summarizes the features of some of the optimizers currently available in HyperSpy, including whether they support parameter bounds, gradients and parameter error estimation. The “Type” column indicates whether the optimizers find a local or global optima.

Table 1: Features of supported curve-fitting optimizers.

Optimizer	Bounds	Gradients	Errors	Loss function	Type	Linear
"lm" (default)	Yes	Yes	Yes	Only "ls"	local	No
"trf"	Yes	Yes	Yes	Only "ls"	local	No
"dogbox"	Yes	Yes	Yes	Only "ls"	local	No
"odr"	No	Yes	Yes	Only "ls"	local	No
"lstsq"	No	No	Yes ¹	Only "ls"	global	Yes
"ridge_regression"	No	No	Yes ^{Page 156, 1}	Only "ls"	global	Yes
<code>scipy.optimize.minimize()</code>	Yes ²	Yes ²	No	All	local	No
"Differential Evolution"	Yes	No	No	All	global	No
"Dual Annealing" ³	Yes	No	No	All	global	No
"SHGO" ³	Yes	No	No	All	global	No

The default optimizer in HyperSpy is "lm", which stands for the [Levenberg-Marquardt algorithm](#). In earlier versions of HyperSpy (< 1.6) this was known as "leastsq".

10.6.2 Loss functions

HyperSpy supports a number of loss functions. The default is "ls", i.e. the least-squares loss. For the vast majority of cases, this loss function is appropriate, and has the additional benefit of supporting parameter error estimation and *goodness-of-fit* testing. However, if your data contains very low counts per pixel, or is corrupted by outliers, the "ML-poisson" and "huber" loss functions may be worth investigating.

Least squares with error estimation

The following example shows how to perform least squares optimization with error estimation. First we create data consisting of a line $y = a \cdot x + b$ with $a = 1$ and $b = 100$, and we then add Gaussian noise to it:

```
>>> s = hs.signals.Signal1D(np.arange(100, 300, dtype='float32'))
>>> s.add_gaussian_noise(std=100)
```

To fit it, we create a model consisting of a *Polynomial* component of order 1 and fit it to the data.

```
>>> m = s.create_model()
>>> line = hs.model.components1D.Polynomial(order=1)
>>> m.append(line)
>>> m.fit()
```

Once the fit is complete, the optimized value of the parameters and their estimated standard deviation are stored in the following line attributes:

```
>>> line.a0.value
0.9924615648843765
>>> line.a1.value
103.67507406125888
```

(continues on next page)

¹ Requires the `multifit()` `calculate_errors = True` argument in most cases. See the documentation below on [linear least square fitting](#) for more info.

² All of the fitting algorithms available in `scipy.optimize.minimize()` are currently supported by HyperSpy; however, only some of them support bounds and/or gradients. For more information, please see the [SciPy documentation](#).

³ Requires `scipy >= 1.2.0`.

(continued from previous page)

```
>>> line.a0.std
0.11771053738516088
>>> line.a1.std
13.541061301257537
```

Warning: When the noise is heteroscedastic, only if the `metadata.Signal.Noise_properties.variance` attribute of the `Signal1D` instance is defined can the parameter standard deviations be estimated accurately.

If the variance is not defined, the standard deviations are still computed, by setting variance equal to 1. However, this calculation will not be correct unless an accurate value of the variance is provided. See [Setting the noise properties](#) for more information.

Weighted least squares with error estimation

In the following example, we add Poisson noise to the data instead of Gaussian noise, and proceed to fit as in the previous example.

```
>>> s = hs.signals.Signal1D(np.arange(300))
>>> s.add_poissonian_noise()
>>> m = s.create_model()
>>> line = hs.model.components1D.Polynomial(order=1)
>>> m.append(line)
>>> m.fit()
>>> line.a0.value
-0.7262000522775925
>>> line.a1.value
1.0086925334859176
>>> line.a0.std
1.4141418570079
>>> line.a1.std
0.008185019194679451
```

Because the noise is heteroscedastic, the least squares optimizer estimation is biased. A more accurate result can be obtained with weighted least squares, where the weights are proportional to the inverse of the noise variance. Although this is still biased for Poisson noise, it is a good approximation in most cases where there are a sufficient number of counts per pixel.

```
>>> exp_val = hs.signals.Signal1D(np.arange(300)+1)
>>> s.estimate_poissonian_noise_variance(expected_value=exp_val)
>>> line.estimate_parameters(s, 10, 250)
True
>>> m.fit()
>>> line.a0.value
-0.6666008600519397
>>> line.a1.value
1.017145603577098
>>> line.a0.std
0.8681360488613021
>>> line.a1.std
0.010308732161043038
```

Warning: When the attribute `metadata.Signal.Noise_properties.variance` is defined, the behaviour is to perform a weighted least-squares fit using the inverse of the noise variance as the weights. In this scenario, to then disable weighting, you will need to **unset** the attribute. You can achieve this with `set_noise_variance()`:

```
>>> m.signal.set_noise_variance(None) # This will now be an unweighted fit
>>> m.fit()
>>> line.a0.value
-1.9711403542163477
>>> line.a1.value
1.0258716193502546
```

Poisson maximum likelihood estimation

To avoid biased estimation in the case of data corrupted by Poisson noise with very few counts, we can use Poisson maximum likelihood estimation (MLE) instead. This is an unbiased estimator for Poisson noise. To perform MLE, we must use a general, non-linear optimizer from the [table above](#), such as Nelder-Mead or L-BFGS-B:

```
>>> m.fit(optimizer="Nelder-Mead", loss_function="ML-poisson")
>>> line.a0.value
0.00025567973144090695
>>> line.a1.value
1.0036866523183754
```

Estimation of the parameter errors is not currently supported for Poisson maximum likelihood estimation.

Huber loss function

HyperSpy also implements the [Huber loss](#) function, which is typically less sensitive to outliers in the data compared to the least-squares loss. Again, we need to use one of the general non-linear optimization algorithms:

```
>>> m.fit(optimizer="Nelder-Mead", loss_function="huber")
```

Estimation of the parameter errors is not currently supported for the Huber loss function.

Custom loss functions

As well as the built-in loss functions described above, a custom loss function can be passed to the model:

```
>>> def my_custom_function(model, values, data, weights=None):
...     """
...     Parameters
...     -----
...     model : Model instance
...         the model that is fitted.
...     values : np.ndarray
...         A one-dimensional array with free parameter values suggested by the
...         optimizer (that are not yet stored in the model).
...     data : np.ndarray
...         A one-dimensional array with current data that is being fitted.
...     weights : {np.ndarray, None}
```

(continues on next page)

(continued from previous page)

```

...     An optional one-dimensional array with parameter weights.
...
... Returns
... -----
... score : float
...     A single float value, representing a score of the fit, with
...     lower values corresponding to better fits.
...     """
...     # Almost any operation can be performed, for example:
...     # First we store the suggested values in the model
...     model.fetch_values_from_array(values)
...
...     # Evaluate the current model
...     cur_value = model(onlyactive=True)
...
...     # Calculate the weighted difference with data
...     if weights is None:
...         weights = 1
...     difference = (data - cur_value) * weights
...
...     # Return squared and summed weighted difference
...     return (difference**2).sum()

>>> # We must use a general non-linear optimizer
>>> m.fit(optimizer='Nelder-Mead', loss_function=my_custom_function)

```

If the optimizer requires an analytical gradient function, it can be similarly passed, using the following signature:

```

>>> def my_custom_gradient_function(model, values, data, weights=None):
...     """
...     Parameters
...     -----
...     model : Model instance
...         the model that is fitted.
...     values : np.ndarray
...         A one-dimensional array with free parameter values suggested by the
...         optimizer (that are not yet stored in the model).
...     data : np.ndarray
...         A one-dimensional array with current data that is being fitted.
...     weights : {np.ndarray, None}
...         An optional one-dimensional array with parameter weights.
...
...     Returns
...     -----
...     gradients : np.ndarray
...         a one-dimensional array of gradients, the size of `values`,
...         containing each parameter gradient with the given values
...     """
...     # As an example, estimate maximum likelihood gradient:
...     model.fetch_values_from_array(values)
...     cur_value = model(onlyactive=True)
...

```

(continues on next page)

(continued from previous page)

```
... # We use in-built jacobian estimation
... jac = model._jacobian(values, data)
...
... return -(jac * (data / cur_value - 1)).sum(1)

>>> # We must use a general non-linear optimizer again
>>> m.fit(optimizer='L-BFGS-B',
...       loss_function=my_custom_function,
...       grad=my_custom_gradient_function)
```

10.6.3 Using gradient information

New in version 1.6: `grad="analytical"` and `grad="fd"` keyword arguments

Optimization algorithms that take into account the gradient of the loss function will often out-perform so-called “derivative-free” optimization algorithms in terms of how rapidly they converge to a solution. HyperSpy can use analytical gradients for model-fitting, as well as numerical estimates of the gradient based on finite differences.

If all the components in the model support analytical gradients, you can pass `grad="analytical"` in order to use this information when fitting. The results are typically more accurate than an estimated gradient, and the optimization often runs faster since fewer function evaluations are required to calculate the gradient.

Following the above examples:

```
>>> m = s.create_model()
>>> line = hs.model.components1D.Polynomial(order=1)
>>> m.append(line)

>>> # Use a 2-point finite-difference scheme to estimate the gradient
>>> m.fit(grad="fd", fd_scheme="2-point")

>>> # Use the analytical gradient
>>> m.fit(grad="analytical")

>>> # Huber loss and Poisson MLE functions
>>> # also support analytical gradients
>>> m.fit(grad="analytical", loss_function="huber")
>>> m.fit(grad="analytical", loss_function="ML-poisson")
```

Note: Analytical gradients are not yet implemented for the `Model2D` class.

10.6.4 Bounded optimization

Non-linear optimization can sometimes fail to converge to a good optimum, especially if poor starting values are provided. Problems of ill-conditioning and non-convergence can be improved by using bounded optimization.

All components' parameters have the attributes `parameter.bmin` and `parameter.bmax` ("bounded min" and "bounded max"). When fitting using the `bounded=True` argument by `m.fit(bounded=True)` or `m.multifit(bounded=True)`, these attributes set the minimum and maximum values allowed for `parameter.value`.

Currently, not all optimizers support bounds - see the [table above](#). In the following example, a Gaussian histogram is fitted using a *Gaussian* component using the Levenberg-Marquardt ("lm") optimizer and bounds on the centre parameter.

```
>>> s = hs.signals.BaseSignal(np.random.normal(loc=10, scale=0.01,
... size=100000)).get_histogram()
>>> s.axes_manager[-1].is_binned = True
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> m.append(g1)
>>> g1.centre.value = 7
>>> g1.centre.bmin = 7
>>> g1.centre.bmax = 14
>>> m.fit(optimizer="lm", bounded=True)
>>> m.print_current_values()
Model1D: histogram
Gaussian: Gaussian
Active: True
Parameter Name | Free | Value | Std | Min | Max
===== | ===== | ===== | ===== | ===== | =====
A | True | 99997.3481 | 232.333693 | 0.0 | None
sigma | True | 0.00999184 | 2.68064163 | None | None
centre | True | 9.99980788 | 2.68064070 | 7.0 | 14.0
```

10.6.5 Linear least squares

New in version 1.7.

Linear fitting can be used to address some of the drawbacks of non-linear optimization:

- it doesn't suffer from the *starting parameters* issue, which can sometimes be problematic with nonlinear fitting. Since linear fitting uses linear algebra to find the solution (find the parameter values of the model), the solution is a unique solution, while nonlinear optimization uses an iterative approach and therefore relies on the initial values of the parameters.
- it is fast, because i) in favorable situations, the signal can be fitted in a vectorized fashion, i.e. the signal is fitted in a single run instead of iterating over the navigation dimension; ii) it is not iterative, i.e. it does the calculation only one time instead of 10-100 iterations, depending on how quickly the non-linear optimizer will converge.

However, linear fitting can *only* fit linear models and will not be able to fit parameters which vary *non-linearly*.

A component is considered linear when its free parameters scale the component only in the y-axis. For the exemplary function $y = a*x**b$, a is a linear parameter, whilst b is not. If `b.free = False`, then the component is linear. Components can also be made up of several linear parts. For instance, the 2D-polynomial $y = a*x**2+b*y**2+c*x+d*y+e$ is entirely linear.

Note: After creating a model with values for the nonlinear parameters, a quick way to set all nonlinear parameters to be free = False is to use `m.set_parameters_not_free(only_nonlinear=True)`

To check if a parameter is linear, use the model or component method `print_current_values()`. For a component to be considered linear, it can hold only one free parameter, and that parameter must be linear.

If all components in a model are linear, then a linear optimizer can be used to solve the problem as a linear regression problem! This can be done using two approaches:

- the standard pixel-by-pixel approach as used by the *nonlinear* optimizers
- fit the entire dataset in one *vectorised* operation, which will be much faster (up to 1000 times). However, there is a caveat: all fixed parameters must have the same value across the dataset in order to avoid creating a very large array whose size will scale with the number of different values of the non-free parameters.

Note: A good example of a linear model in the electron-microscopy field is an Energy-Dispersive X-ray Spectroscopy (EDS) dataset, which typically consists of a polynomial background and Gaussian peaks with well-defined energy (Gaussian.centre) and peak widths (Gaussian.sigma). This dataset can be fit extremely fast with a linear optimizer.

There are two implementations of linear least squares fitting in hyperspy:

- the 'lstsq' optimizer, which uses `numpy.linalg.lstsq()`, or `dask.array.linalg.lstsq()` for lazy signals.
- the 'ridge_regression' optimizer, which supports regularization (see `sklearn.linear_model.Ridge` for arguments to pass to `fit()`), but does not support lazy signals.

As for non-linear least squares fitting, *weighted least squares* is supported.

In the following example, we first generate a 300x300 navigation signal of varying total intensity, and then populate it with an EDS spectrum at each point. The signal can be fitted with a polynomial background and a Gaussian for each peak. Hyperspy automatically adds these to the model, and fixes the `centre` and `sigma` parameters to known values. Fitting this model with a non-linear optimizer can about half an hour on a decent workstation. With a linear optimizer, it takes seconds.

```
>>> nav = hs.signals.Signal2D(np.random.random((300, 300))).T
>>> s = exspy.data.EDS_TEM_FePt_nanoparticles() * nav
>>> m = s.create_model()

>>> m.multifit(optimizer='lstsq')
```

Standard errors for the parameters are by default not calculated when the dataset is fitted in vectorized fashion, because it has large memory requirement. If errors are required, either pass `calculate_errors=True` as an argument to `multifit()`, or rerun `multifit()` with a nonlinear optimizer, which should run fast since the parameters are already optimized.

None of the linear optimizers currently support bounds.

10.6.6 Optimization results

After fitting the model, details about the optimization procedure, including whether it finished successfully, are returned as `scipy.optimize.OptimizeResult` object, according to the keyword argument `return_info=True`. These details are often useful for diagnosing problems such as a poorly-fitted model or a convergence failure. You can also access the object as the `fit_output` attribute:

```
>>> m.fit()
>>> type(m.fit_output)
<scipy.optimize.OptimizeResult object>
```

You can also print this information using the `print_info` keyword argument:

```
# Print the info to stdout
>>> m.fit(optimizer="L-BFGS-B", print_info=True) # doctest: +SKIP
Fit info:
  optimizer=L-BFGS-B
  loss_function=ls
  bounded=False
  grad="fd"
Fit result:
  hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<= _FACTR*EPSMCH'
  nfev: 168
  nit: 32
  njev: 42
  status: 0
  success: True
  x: array([ 9.97614503e+03, -1.10610734e-01,  1.98380701e+00])
```

10.6.7 Goodness of fit

The chi-squared, reduced chi-squared and the degrees of freedom are computed automatically when fitting a (weighted) least-squares model (i.e. only when `loss_function="ls"`). They are stored as signals, in the `chisq`, `red_chisq` and `dof` attributes of the model respectively.

Warning: Unless `metadata.Signal.Noise_properties.variance` contains an accurate estimation of the variance of the data, the chi-squared and reduced chi-squared will not be computed correctly. This is true for both homocedastic and heteroscedastic noise.

10.6.8 Visualizing the model

To visualise the result use the `plot()` method:

```
>>> m.plot() # Visualise the results
```

By default only the full model line is displayed in the plot. In addition, it is possible to display the individual components by calling `enable_plot_components()` or directly using `plot()`:

```
>>> m.plot(plot_components=True) # Visualise the results
```

To disable this feature call `disable_plot_components()`.

New in version 1.4: `plot()` keyword arguments

All extra keyword arguments are passed to the `plot()` method of the corresponding signal object. The following example plots the model signal figure but not its navigator:

```
>>> m.plot(navigator=False)
```

By default the model plot is automatically updated when any parameter value changes. It is possible to suspend this feature with `suspend_update()`.

10.6.9 Setting the initial parameters

Non-linear optimization often requires setting sensible starting parameters. This can be done by plotting the model and adjusting the parameters by hand.

Changed in version 1.3: All `notebook_interaction` methods renamed to `gui()`. The `notebook_interaction` methods were removed in 2.0. If running in a Jupyter Notebook, interactive widgets can be used to conveniently adjust the parameter values by running `gui()` for `BaseModel`, `Component` and `Parameter`.

```
m = s.create_model()
m.append(hs.model.components.Gaussian())
m.append(hs.model.components.Polynomial(5))
m.append(hs.model.components.EELSCLEdge('C_K'))
m.append(hs.model.components.ScalableFixedPattern(s.deepcopy()))
m.notebook_interaction()
```

The screenshot shows a Jupyter notebook interface with a code cell and a widget panel. The code cell contains the following Python code:

```
m = s.create_model()
m.append(hs.model.components.Gaussian())
m.append(hs.model.components.Polynomial(5))
m.append(hs.model.components.EELSCLEdge('C_K'))
m.append(hs.model.components.ScalableFixedPattern(s.deepcopy()))
m.notebook_interaction()
```

The widget panel displays the following components:

- background**: A button to toggle the background.
- Gaussian**: A widget for the Gaussian component.
- Polynomial**: A widget for the Polynomial component.
- C_K**: A widget for the EELSCLEdge component.
- ScalableFixedPattern**: A widget for the ScalableFixedPattern component, which includes:
 - active
 - interpolate
 - yscale**: A slider with a minimum value of -9 and a maximum value of 11.
 - xscale**: A slider with a minimum value of -9 and a maximum value of 11.
 - shift**: A slider with a minimum value of 0 and a maximum value of 499.

Fig. 1: Interactive widgets for the full model in a Jupyter notebook. Drag the sliders to adjust current parameter values. Typing different minimum and maximum values changes the boundaries of the slider.

Also, `enable_adjust_position()` provides an interactive way of setting the position of the components with a well-defined position. `disable_adjust_position()` disables the tool.

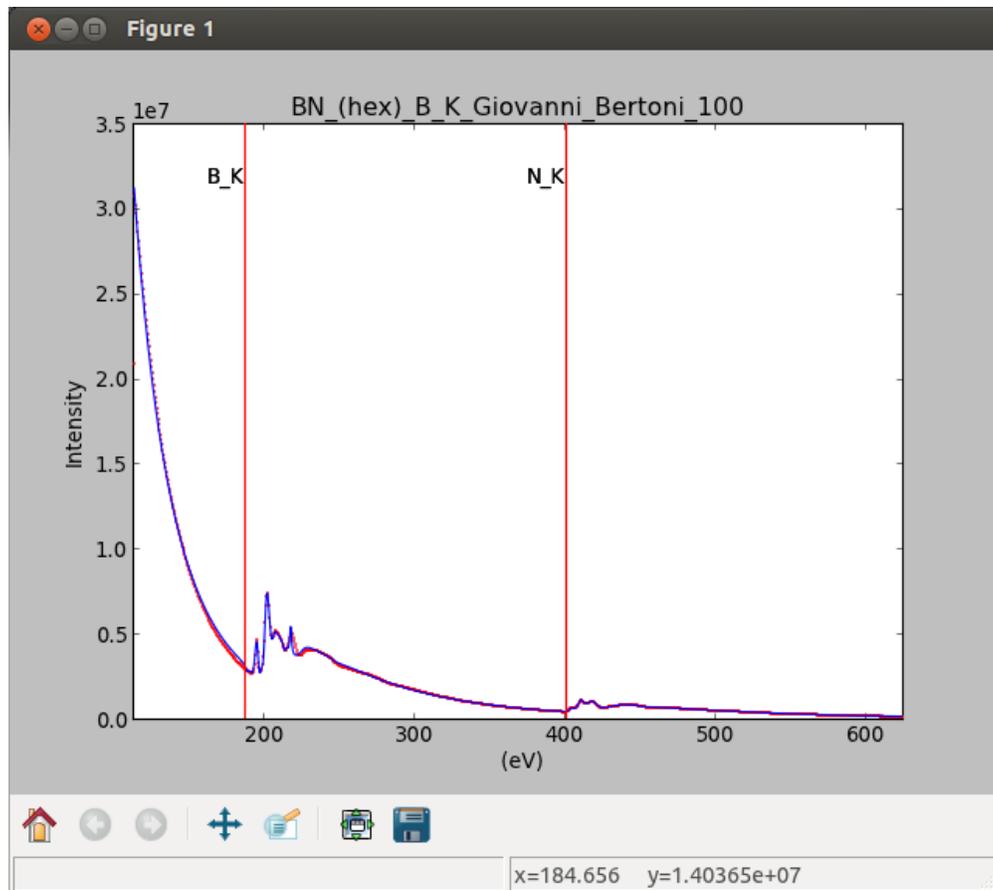


Fig. 2: Interactive component position adjustment tool. Drag the vertical lines to set the initial value of the position parameter.

10.6.10 Exclude data from the fitting process

The following *BaseModel* methods can be used to exclude undesired spectral channels from the fitting process:

- `set_signal_range()`
- `add_signal_range()`
- `set_signal_range_from_mask()`
- `remove_signal_range()`
- `reset_signal_range()`

The example below shows how a boolean array can be easily created from the signal and how the `isig` syntax can be used to define the signal range.

```
>>> # Create a sample 2D gaussian dataset
>>> g = hs.model.components2D.Gaussian2D(
...     A=1, centre_x=-5.0, centre_y=-5.0, sigma_x=1.0, sigma_y=2.0,)

>>> scale = 0.1
>>> x = np.arange(-10, 10, scale)
>>> y = np.arange(-10, 10, scale)
>>> X, Y = np.meshgrid(x, y)

>>> im = hs.signals.Signal2D(g.function(X, Y))
>>> im.axes_manager[0].scale = scale
>>> im.axes_manager[0].offset = -10
>>> im.axes_manager[1].scale = scale
>>> im.axes_manager[1].offset = -10

>>> m = im.create_model() # Model initialisation
>>> gt = hs.model.components2D.Gaussian2D()
>>> m.append(gt)

>>> m.set_signal_range(-7, -3, -9, -1) # Set signal range
>>> m.fit()
```

Alternatively, create a boolean signal of the same shape as the signal space of `im`

```
>>> signal_mask = im > 0.01

>>> m.set_signal_range_from_mask(signal_mask.data) # Set signal range
>>> m.fit()
```

10.6.11 Fitting multidimensional datasets

To fit the model to all the elements of a multidimensional dataset, use `multifit()`:

```
>>> m.multifit() # warning: this can be a lengthy process on large datasets
```

`multifit()` fits the model at the first position, stores the result of the fit internally and move to the next position until reaching the end of the dataset.

Note: Sometimes this method can fail, especially in the case of a TEM spectrum image of a particle surrounded by vacuum (since in that case the top-left pixel will typically be an empty signal).

To get sensible starting parameters, you can do a single `fit()` after changing the active position within the spectrum image (either using the plotting GUI or by directly modifying `s.axes_manager.indices` as in *Setting axis properties*).

After doing this, you can initialize the model at every pixel to the values from the single pixel fit using `m.assign_current_values_to_all()`, and then use `multifit()` to perform the fit over the entire spectrum image.

New in version 1.6: New optional fitting iteration path “*serpentine*”

New in version 2.0: New default iteration path for fitting is “serpentine”

In HyperSpy, curve fitting on a multidimensional dataset happens in the following manner: Pixels are fit along the row from the first index in the first row, and once the last pixel in the row is reached, one proceeds in reverse order from the last index in the second row. This procedure leads to a serpentine pattern, as seen on the image below. The serpentine pattern supports n-dimensional navigation space, so the first index in the second frame of a three-dimensional navigation space will be at the last position of the previous frame.

An alternative scan pattern would be the 'flyback' scheme, where the map is iterated through row by row, always starting from the first index. This pattern can be explicitly set using the `multifit() iterpath='flyback'` argument. However, the 'serpentine' strategy is usually more robust, as it always moves on to a neighbouring pixel and the fitting procedure uses the fit result of the previous pixel as the starting point for the next. A common problem in the 'flyback' pattern is that the fitting fails going from the end of one row to the beginning of the next, as the spectrum can change abruptly.

In addition to 'serpentine' and 'flyback', `iterpath` can take as argument any list or array of indices, or a generator of such, as explained in the *Iterating AxesManager* section.

Sometimes one may like to store and fetch the value of the parameters at a given position manually. This is possible using `store_current_values()` and `fetch_stored_values()`.

10.6.12 Visualising the result of the fit

The `BaseModel.plot_results()`, `Component.plot()` and `Parameter.plot()` methods can be used to visualise the result of the fit **when fitting multidimensional datasets**.

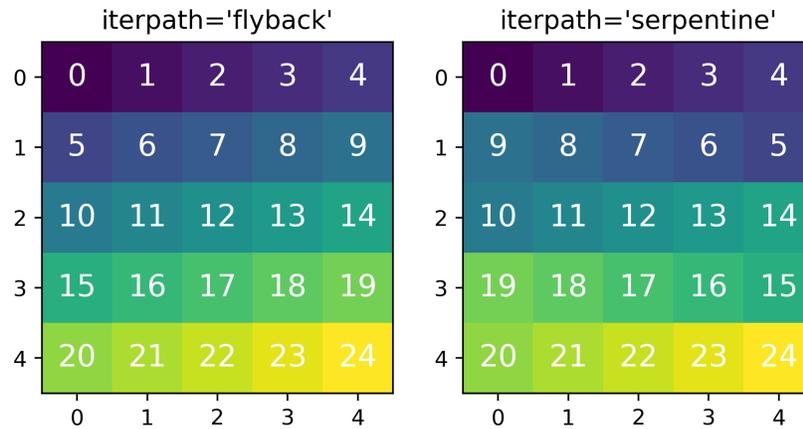


Fig. 3: Comparing the scan patterns generated by the 'flyback' and 'serpentine' iterpath options for a 2D navigation space. The pixel intensity and number refers to the order that the signal is fitted in.

10.7 Storing models

Multiple models can be stored in the same signal. In particular, when `store()` is called, a full “frozen” copy of the model is stored in the signal’s *ModelManager*, which can be accessed in the `models` attribute (i.e. `s.models`). The stored models can be recreated at any time by calling `restore()` with the stored model name as an argument. To remove a model from storage, simply call `remove()`.

The stored models can be either given a name, or assigned one automatically. The automatic naming follows alphabetical scheme, with the sequence being (a, b, ..., z, aa, ab, ..., az, ba, ...).

Note: If you want to slice a model, you have to perform the operation on the model itself, not its stored version

Warning: Modifying a signal in-place (e.g. `map()`, `crop()`, `align1D()`, `align2D()` and similar) will invalidate all stored models. This is done intentionally.

Current stored models can be listed by calling `s.models`:

```
>>> s = hs.signals.Signal1D(np.arange(100))
>>> m = s.create_model()
>>> m.append(hs.model.components1D.Lorentzian())
>>> m.store('myname')
>>> s.models
```

(continues on next page)

(continued from previous page)

```

└─ myname
  ├── components
  │   └─ Lorentzian
  ├── date = 2015-09-07 12:01:50
  └─ dimensions = (|100)

>>> m.append(hs.model.components1D.Exponential())
>>> m.store() # assign model name automatically
>>> s.models
└─ a
  ├── components
  │   ├── Exponential
  │   └─ Lorentzian
  ├── date = 2015-09-07 12:01:57
  └─ dimensions = (|100)
└─ myname
  ├── components
  │   └─ Lorentzian
  ├── date = 2015-09-07 12:01:50
  └─ dimensions = (|100)

>>> m1 = s.models.restore('myname')
>>> m1.components
# | Attribute Name | Component Name | Component Type
---|-----|-----|-----
0 | Lorentzian | Lorentzian | Lorentzian

```

10.7.1 Saving and loading the result of the fit

To save a model, a convenience function `save()` is provided, which stores the current model into its signal and saves the signal. As described in *Storing models*, more than just one model can be saved with one signal.

```

>>> m = s.create_model()
>>> # analysis and fitting goes here
>>> m.save('my_filename', 'model_name')
>>> l = hs.load('my_filename.hspy')
>>> m = l.models.restore('model_name')

```

Alternatively

```

>>> m = l.models.model_name.restore()

```

For older versions of HyperSpy (before 0.9), the instructions were as follows:

Note that this method is known to be brittle i.e. there is no guarantee that a version of HyperSpy different from the one used to save the model will be able to load it successfully. Also, it is advisable not to use this method in combination with functions that alter the value of the parameters interactively (e.g. `enable_adjust_position`) as the modifications made by this functions are normally not stored in the IPython notebook or Python script.

To save a model:

1. Save the parameter arrays to a file using `save_parameters2file()`.

2. Save all the commands that used to create the model to a file. This can be done in the form of an IPython notebook or a Python script.
3. (Optional) Comment out or delete the fitting commands (e.g. `multifit()`).

To recreate the model:

1. Execute the IPython notebook or Python script.
2. Use `load_parameters_from_file()` to load back the parameter values and arrays.

10.7.2 Exporting the result of the fit

The `BaseModel export_results()`, `Component export()` and `Parameter export()` methods can be used to export the result of the optimization in all supported formats.

10.8 Fitting big data

See the section in *Model fitting* working with big data.

10.9 Smart Adaptive Multi-dimensional Fitting (SAMFire)

SAMFire (Smart Adaptive Multi-dimensional Fitting) is an algorithm created to reduce the starting value (or local / false minima) problem, which often arises when fitting multi-dimensional datasets.

The algorithm is described in [Tomas Ostasevicius' PhD thesis](#), entitled “Multi-dimensional Data Analysis in Electron Microscopy”.

10.9.1 The idea

The main idea of SAMFire is to change two things compared to the traditional way of fitting datasets with many dimensions in the navigation space:

1. Pick a more sensible pixel fitting order.
2. Calculate the pixel starting parameters from already fitted parts of the dataset.

Both of these aspects are linked one to another and are represented by two different strategy families that SAMFire uses while operating.

10.9.2 Strategies

During operation SAMFire uses a list of strategies to determine how to select the next pixel and estimate its starting parameters. Only one strategy is used at a time. Next strategy is chosen when no new pixels can be fitted with the current strategy. Once either the strategy list is exhausted or the full dataset fitted, the algorithm terminates.

There are two families of strategies. In each family there may be many strategies, using different statistical or significance measures.

As a rule of thumb, the first strategy in the list should always be from the local family, followed by a strategy from the global family.

10.9.3 Local strategy family

These strategies assume that locally neighbouring pixels are similar. As a result, the pixel fitting order seems to follow data-suggested order, and the starting values are computed from the surrounding already fitted pixels.

More information about the exact procedure will be available once the accompanying paper is published.

10.9.4 Global strategy family

Global strategies assume that the navigation coordinates of each pixel bear no relation to its signal (i.e. the location of pixels is meaningless). As a result, the pixels are selected at random to ensure uniform sampling of the navigation space.

A number of candidate starting values are computed from global statistical measures. These values are all attempted in order until a satisfactory result is found (not necessarily testing all available starting guesses). As a result, on average each pixel requires significantly more computations when compared to a local strategy.

More information about the exact procedure will be available once the accompanying paper is published.

10.9.5 Seed points

Due to the strategies using already fitted pixels to estimate the starting values, at least one pixel has to be fitted beforehand by the user.

The seed pixel(s) should be selected to require the most complex model present in the dataset, however in-built goodness of fit checks ensure that only sufficiently well fitted values are allowed to propagate.

If the dataset consists of regions (in the navigation space) of highly dissimilar pixels, often called “domain structures”, at least one seed pixel should be given for each unique region.

If the starting pixels were not optimal, only part of the dataset will be fitted. In such cases it is best to allow the algorithm terminate, then provide new (better) seed pixels by hand, and restart SAMFire. It will use the new seed together with the already computed parts of the data.

10.9.6 Usage

After creating a model and fitting suitable seed pixels, to fit the rest of the multi-dimensional dataset using SAMFire we must create a SAMFire instance as follows:

```
>>> samf = m.create_samfire(workers=None, ipyparallel=False)
```

By default SAMFire will look for an `ipyparallel` cluster for the workers for around 30 seconds. If none is available, it will use multiprocessing instead. However, if you are not planning to use `ipyparallel`, it's recommended specify it explicitly via the `ipyparallel=False` argument, to use the fall-back option of *multiprocessing*.

By default a new SAMFire object already has two (and currently only) strategies added to its `strategies` list:

```
>>> samf.strategies
A | # | Strategy
--|---|-----
x | 0 | Reduced chi squared strategy
  | 1 | Histogram global strategy
```

The currently active strategy is marked by an ‘x’ in the first column.

If a new datapoint (i.e. pixel) is added manually, the “database” of the currently active strategy has to be refreshed using the `refresh_database()` call.

The current strategy “database” can be plotted using the `plot()` method.

Whilst SAMFire is running, each pixel is checked by a `goodness_test`, which is by default `red_chisq_test`, checking the reduced chi-squared to be in the bounds of [0, 2].

This tolerance can (and most likely should!) be changed appropriately for the data as follows:

```
>>> # use a sensible value
>>> samf.metadata.goodness_test.tolerance = 0.3
```

The SAMFire managed multi-dimensional fit can be started using the `start()` method. All keyword arguments are passed to the underlying (i.e. usual) `fit()` call:

```
>>> samf.start(optimizer='lm', bounded=True)
```

WORKING WITH BIG DATA

New in version 1.2.

HyperSpy makes it possible to analyse data larger than the available memory by providing “lazy” versions of most of its signals and functions. In most cases the syntax remains the same. This chapter describes how to work with data larger than memory using the *LazySignal* class and its derivatives.

11.1 Creating Lazy Signals

11.1.1 Lazy Signals from external data

If the data is large and not loaded by HyperSpy (for example a `hdf5.Dataset` or similar), first wrap it in `dask.array.Array` as shown [here](#) and then pass it as normal and call `as_lazy()`:

```
>>> import h5py
>>> f = h5py.File("myfile.hdf5")
>>> data = f['/data/path']

Wrap the data in dask and chunk as appropriate

>>> import dask.array as da
>>> x = da.from_array(data, chunks=(1000, 100))

Create the lazy signal

>>> s = hs.signals.Signal1D(x).as_lazy()
```

11.1.2 Loading lazily

To load the data lazily, pass the keyword `lazy=True`. As an example, loading a 34.9 GB `.blo` file on a regular laptop might look like:

```
>>> s = hs.load("shish26.02-6.blo", lazy=True)
>>> s
<LazySignal2D, title: , dimensions: (400, 333|512, 512)>
>>> s.data
dask.array<array-e..., shape=(333, 400, 512, 512), dtype=uint8, chunksize=(20, 12, 512, ↵
↵512)>
>>> print(s.data.dtype, s.data.nbytes / 1e9)
```

(continues on next page)

(continued from previous page)

```
uint8 34.9175808
```

Change dtype to perform decomposition, etc.

```
>>> s.change_dtype("float")
>>> print(s.data.dtype, s.data.nbytes / 1e9)
float64 279.3406464
```

Loading the dataset in the original unsigned integer format would require around 35GB of memory. To store it in a floating-point format one would need almost 280GB of memory. However, with the lazy processing both of these steps are near-instantaneous and require very little computational resources.

New in version 1.4: `close_file()`

Currently when loading an hdf5 file lazily the file remains open at least while the signal exists. In order to close it explicitly, use the `close_file()` method. Alternatively, you could close it on calling `compute()` by passing the keyword argument `close_file=True` e.g.:

```
>>> s = hs.load("file.hspy", lazy=True)
>>> ssum = s.sum(axis=0)
```

Close the file

```
>>> ssum.compute(close_file=True)
```

11.1.3 Lazy stacking

Occasionally the full dataset consists of many smaller files. To combine them into a one large LazySignal, we can *stack* them lazily (both when loading or afterwards):

```
>>> siglist = hs.load("*.hdf5")
>>> s = hs.stack(siglist, lazy=True)
```

Or load lazily and stack afterwards:

```
>>> siglist = hs.load("*.hdf5", lazy=True)
```

Make a stack, no need to pass 'lazy', as signals are already lazy

```
>>> s = hs.stack(siglist)
```

Or do everything in one go:

```
>>> s = hs.load("*.hdf5", lazy=True, stack=True)
```

11.1.4 Casting signals as lazy

To convert a regular HyperSpy signal to a lazy one such that any future operations are only performed lazily, use the `as_lazy()` method:

```
>>> s = hs.signals.Signal1D(np.arange(150.).reshape((3, 50)))
>>> s
<Signal1D, title: , dimensions: (3|50)>
>>> sl = s.as_lazy()
>>> sl
<LazySignal1D, title: , dimensions: (3|50)>
```

11.2 Machine learning

Warning: The machine learning features are in beta state.

Although most of them work as described, their operation may not always be optimal, well-documented and/or consistent with their in-memory counterparts.

Decomposition algorithms for machine learning often perform large matrix manipulations, requiring significantly more memory than the data size. To perform decomposition operation lazily, HyperSpy provides access to several “online” algorithms as well as `dask`’s lazy SVD algorithm. Online algorithms perform the decomposition by operating serially on chunks of data, enabling the lazy decomposition of large datasets. In line with the standard HyperSpy signals, `decomposition()` offers the following online algorithms:

Table 1: Available lazy decomposition algorithms in HyperSpy

Algorithm	Method
“SVD” (default)	<code>dask.array.linalg.svd()</code>
“PCA”	<code>sklearn.decomposition.IncrementalPCA</code>
“ORPCA”	<code>ORPCA</code>
“ORNMF”	<code>ORNMF</code>

See also:

`decomposition()` for more details on decomposition with non-lazy signals.

11.3 Navigator plot

The default signal navigator is the sum of the signal across all signal dimensions and all but 1 or 2 navigation dimensions. If the dataset is large, this can take a significant amount of time to perform with every plot. By default, a navigator is computed with minimally required approach to obtain a good signal-to-noise ratio image: the sum is taken on a single chunk of the signal space, in order to avoid to compute the navigator for the whole dataset. In the following example, the signal space is divided in 25 chunks (5 along on each axis), and therefore computing the navigation will only be performed over a small subset of the whole dataset by taking the sum on only 1 chunk out of 25:

```
>>> import dask.array as da
>>> import hyperspy.api as hs
```

(continues on next page)

(continued from previous page)

```
>>> data = da.random.random((100, 100, 1000, 1000), chunks=('auto', 'auto', 200, 200))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s.plot()
```

In the example above, the calculation of the navigation is fast but the actual visualisation of the dataset is slow, each for each navigation index change, 25 chunks of the dataset needs to be fetched from the harddrive. In the following example, the signal space contains a single chunk (instead of 25, in the previous example) and the calculating the navigator will then be slower (~20x) because the whole dataset will need to be processed, however in this case, the visualisation will be faster, because only a single chunk will be fetched from the harddrive when changing navigation indices:

```
>>> data = da.random.random((100, 100, 1000, 1000), chunks=('auto', 'auto', 1000, 1000))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s.plot()
```

This approach depends heavily on the chunking of the data and may not be always suitable. The `compute_navigator()` can be used to calculate the navigator efficiently and store the navigator, so that it can be used when plotting and saved for the later loading of the dataset. The `compute_navigator()` has optional arguments to specify the index where the sum needs to be calculated and how to rechunk the dataset when calculating the navigator. This allows to efficiently calculate the navigator without changing the actual chunking of the dataset, since the rechunking only takes place during the computation of the navigator:

```
>>> data = da.random.random((100, 100, 1000, 1000), chunks=('auto', 'auto', 100, 100))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s.compute_navigator(chunks_number=5)
>>> s.plot()
```

```
>>> data = da.random.random((100, 100, 2000, 400), chunks=('auto', 'auto', 100, 100))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s
<LazySignal2D, title: , dimensions: (100, 100|400, 2000)>
>>> s.compute_navigator(chunks_number=(2, 10))
>>> s.plot()
>>> s.navigator.original_metadata
└─ sum_from = [slice(200, 400, None), slice(1000, 1200, None)]
```

The index can also be specified following the *HyperSpy indexing signal1D* syntax for float and integer.

```
>>> data = da.random.random((100, 100, 2000, 400), chunks=('auto', 'auto', 100, 100))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s
<LazySignal2D, title: , dimensions: (100, 100|400, 2000)>
>>> s.compute_navigator(index=0, chunks_number=(2, 10))
>>> s.navigator.original_metadata
└─ sum_from = [slice(0, 200, None), slice(0, 200, None)]
```

An alternative is to calculate the navigator separately and store it in the signal using the navigator setter.

```
>>> data = da.random.random((100, 100, 1000, 1000), chunks=('auto', 'auto', 100, 100))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s
<LazySignal2D, title: , dimensions: (100, 100|1000, 1000)>
```

For fastest results, just pick one signal space pixel

```
>>> nav = s.isig[500, 500]
```

Alternatively, sum as per default behaviour of non-lazy signal

```
>>> nav = s.sum(s.axes_manager.signal_axes)
>>> nav
<LazySignal2D, title: , dimensions: (|100, 100)>
>>> nav.compute()
[#####] | 100% Completed | 13.1s
>>> s.navigators = nav
>>> s.plot()
```

Alternatively, it is possible to not have a navigator, and use sliders instead

```
>>> s
<LazySignal2D, title: , dimensions: (100, 100|1000, 1000)>
>>> s.plot(navigator='slider')
```

New in version 1.7.

11.4 GPU support

Lazy data processing on GPUs requires explicitly transferring the data to the GPU.

On linux, it is recommended to use the `dask_cuda` library (not supported on windows) to manage the dask scheduler. As for CPU lazy processing, if the dask scheduler is not specified, the default scheduler will be used.

```
>>> from dask_cuda import LocalCUDACluster
>>> from dask.distributed import Client
>>> cluster = LocalCUDACluster()
>>> client = Client(cluster)
```

```
>>> import cupy as cp
>>> import dask.array as da
```

Create a dask array

```
>>> data = da.random.random(size=(20, 20, 100, 100))
>>> data
dask.array<random_sample, shape=(20, 20, 100, 100), dtype=float64, chunksize=(20, 20, ↵
↵100, 100), chunktype=numpy.ndarray>
```

Convert the dask chunks from numpy array to cupy array

```
>>> data = data.map_blocks(cp.asarray)
>>> data
dask.array<random_sample, shape=(20, 20, 100, 100), dtype=float64, chunksize=(20, 20, ↵
↵100, 100), chunktype=cupy.ndarray>
```

Create the signal

```
>>> s = hs.signals.Signal2D(data).as_lazy()
```

Note: See the dask blog on [Richardson Lucy \(RL\) deconvolution](#) for an example of lazy processing on GPUs using dask and cupy

11.5 Model fitting

Most curve-fitting functionality will automatically work on models created from lazily loaded signals. HyperSpy extracts the relevant chunk from the signal and fits to that.

The linear 'lstsq' optimizer supports fitting the entire dataset in a vectorised manner using `dask.array.linalg.lstsq()`. This can give potentially enormous performance benefits over fitting with a nonlinear optimizer, but comes with the restrictions explained in the *linear fitting* section.

11.6 Practical tips

Despite the limitations detailed below, most HyperSpy operations can be performed lazily. Important points are:

- *Chunking*
- *Computing lazy signals*
- *Lazy operations that affect the axes*

11.6.1 Chunking

Data saved in the HDF5 format is typically divided into smaller chunks which can be loaded separately into memory, allowing lazy loading. Chunk size can dramatically affect the speed of various HyperSpy algorithms, so chunk size is worth careful consideration when saving a signal. HyperSpy's default chunking sizes are probably not optimal for a given data analysis technique. For more comprehensible documentation on chunking, see the [dask array chunks](#) and [best practices](#) docs. The chunks saved into HDF5 will match the dask array chunks in `s.data.chunks` when lazy loading. Chunk shape should follow the axes order of the numpy shape (`s.data.shape`), not the hyperspy shape. The following example shows how to chunk one of the two navigation dimensions into smaller chunks:

```
>>> import dask.array as da
>>> data = da.random.random((10, 200, 300))
>>> data.chunksize
(10, 200, 300)

>>> s = hs.signals.Signal1D(data).as_lazy()

Note the reversed order of navigation dimensions

>>> s
<LazySignal1D, title: , dimensions: (200, 10|300)>

Save data with chunking first hyperspy dimension (second array dimension)

>>> s.save('chunked_signal.zspy', chunks=(10, 100, 300))
>>> s2 = hs.load('chunked_signal.zspy', lazy=True)
>>> s2.data.chunksize
(10, 100, 300)
```

To get the chunk size of given axes, the `get_chunk_size()` method can be used:

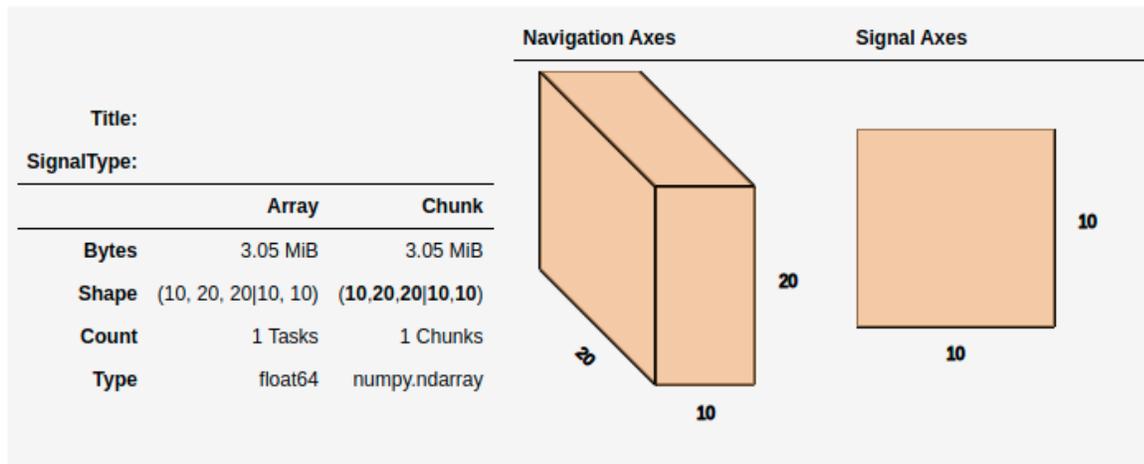
```
>>> import dask.array as da
>>> data = da.random.random((10, 200, 300))
>>> data.chunksize
(10, 200, 300)
>>> s = hs.signals.Signal1D(data).as_lazy()
>>> s.get_chunk_size() # All navigation axes
((10,), (200,))
>>> s.get_chunk_size(0) # The first navigation axis
((200,),)
```

New in version 2.0.0.

Starting in version 2.0.0 HyperSpy does not automatically rechunk datasets as this can lead to reduced performance. The `rechunk` or `optimize` keyword argument can be set to `True` to let HyperSpy automatically change the chunking which could potentially speed up operations.

New in version 1.7.0. For more recent versions of dask (dask>2021.11) when using hyperspy in a jupyter notebook a helpful html representation is available.

```
>>> import dask.array as da
>>> data = da.zeros((20, 20, 10, 10, 10))
>>> s = hs.signals.Signal2D(data).as_lazy()
>>> s
```



This helps to visualize the chunk structure and identify axes where the chunk spans the entire axis (bolded axes).

11.6.2 Computing lazy signals

Upon saving lazy signals, the result of computations is stored on disk.

In order to store the lazy signal in memory (i.e. make it a normal HyperSpy signal) it has a `compute()` method:

```
>>> s
<LazySignal2D, title: , dimensions: (10, 20, 20|10, 10)>
>>> s.compute()
[#####] | 100% Completed | 0.1s
```

(continues on next page)

(continued from previous page)

```
>>> s
<Signal2D, title: , dimensions: (10, 20, 20|10, 10)>
```

11.6.3 Lazy operations that affect the axes

When using lazy signals the computation of the data is delayed until requested. However, the changes to the axes properties are performed when running a given function that modifies them i.e. they are not performed lazily. This can lead to hard to debug issues when the result of a given function that is computed lazily depends on the value of the axes parameters that *may have changed* before the computation is requested. Therefore, in order to avoid such issues, it is recommended to explicitly compute the result of all functions that are affected by the axes parameters. This is the reason why e.g. the result of `shift1D()` is not lazy.

11.7 Dask Backends

Dask is a flexible library for parallel computing in Python. All of the lazy operations in hyperspy run through dask. Dask can be used to run computations on a single machine or scaled to a cluster. The following example shows how to use dask to run computations on a variety of different hardware:

11.7.1 Single Threaded Scheduler

The single threaded scheduler in dask is useful for debugging and testing. It is not recommended for general use.

```
>>> import dask
>>> import hyperspy.api as hs
>>> import numpy as np
>>> import dask.array as da
```

Set the scheduler to single-threaded globally

```
>>> dask.config.set(scheduler='single-threaded')
```

Alternatively, you can set the scheduler to single-threaded for a single function call by setting the `scheduler` keyword argument to `'single-threaded'`.

Or for something like plotting you can set the scheduler to single-threaded for the duration of the plotting call by using the `with dask.config.set` context manager.

```
>>> s.compute(scheduler="single-threaded")

>>> with dask.config.set(scheduler='single-threaded'):
...     s.plot()
```

11.7.2 Single Machine Schedulers

Dask has two schedulers available for single machines.

1. **Threaded Scheduler:**

Fastest to set up but only provides parallelism through threads so only non python functions will be parallelized. This is good if you have largely numpy code and not too many cores.

2. **Processes Scheduler:**

Each task (and all of the necessary dependencies) are shipped to different processes. As such it has a larger set up time. This performs well for python dominated code.

```
>>> import dask
>>> dask.config.set(scheduler='processes')
```

Any hyperspy code will now use the multiprocessing scheduler

```
>>> s.compute()
```

Change to threaded Scheduler, overwrite default

```
>>> dask.config.set(scheduler='threads')
>>> s.compute()
```

11.7.3 Distributed Scheduler

Warning: Distributed computing is not supported for all file formats.

Distributed computing is limited to a few file formats, see the list of [supported file format](#) in RosettaSciIO documentation.

The recommended way to use dask is with the distributed scheduler. This allows you to scale your computations to a cluster of machines. The distributed scheduler can be used on a single machine as well. `dask-distributed` also gives you access to the dask dashboard which allows you to monitor your computations.

Some operations such as the matrix decomposition algorithms in hyperspy don't currently work with the distributed scheduler.

```
>>> from dask.distributed import Client
>>> from dask.distributed import LocalCluster
>>> import dask.array as da
>>> import hyperspy.api as hs
```

```
>>> cluster = LocalCluster()
>>> client = Client(cluster)
>>> client
```

Any calculation will now use the distributed scheduler

```
>>> s
>>> s.plot()
>>> s.compute()
```

Running computation on remote cluster can be done easily using `dask_jobqueue`

```
>>> from dask_jobqueue import SLURMCluster
>>> from dask.distributed import Client
>>> cluster = SLURMCluster(cores=48,
...                       memory='120Gb',
...                       walltime="01:00:00",
...                       queue='research')

Get 3 nodes

>>> cluster.scale(jobs=3)
>>> client = Client(cluster)
>>> client
```

Any calculation will now use the distributed scheduler

```
>>> s = hs.data.two_gaussians()
>>> repeated_data = da.repeat(da.array(s.data[np.newaxis, :]),10, axis=0)
>>> s = hs.signals.Signal1D(repeated_data).as_lazy()
>>> summed = s.map(np.sum, inplace=False)
>>> s.compute()
```

11.8 Limitations

Most operations can be performed lazily. However, lazy operations come with a few limitations and constraints that we detail below.

11.8.1 Immutable signals

An important limitation when using `LazySignal` is the inability to modify existing data (immutability). This is a logical consequence of the DAG (tree structure, explained in *Behind the scenes – technical details*), where a complete history of the processing has to be stored to traverse later.

In fact, lazy evaluation removes the need for such operation, since only additional tree branches are added, requiring very little resources. In practical terms the following fails with lazy signals:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s += 1
Traceback (most recent call last):
  File "<ipython-input-6-1bd1db4187be>", line 1, in <module>
    s += 1
  File "<string>", line 2, in __iadd__
  File "/home/fjd29/Python/hyperspy3/hyperspy/signal.py", line 1591, in _binary_operator_
    ↪ruler
    getattr(self.data, op_name)(other)
AttributeError: 'Array' object has no attribute '__iadd__'
```

However, when operating lazily there is no clear benefit to using in-place operations. So, the operation above could be rewritten as follows:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s = s + 1
```

Or even better:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s1 = s + 1
```

11.8.2 Other minor differences

- **Histograms** for a `LazySignal` do not support `knuth` and `blocks` binning algorithms.
- **CircleROI** sets the elements outside the ROI to `np.nan` instead of using a masked array, because `dask` does not support masking. As a convenience, `nansum`, `nanmean` and other `nan*` signal methods were added to mimic the workflow as closely as possible.

11.8.3 Saving Big Data

The most efficient format supported by HyperSpy to write data is the `ZSpy` format, mainly because it supports writing concurrently from multiple threads or processes. This also allows for smooth interaction with `dask`-distributed for efficient scaling.

11.9 Behind the scenes – technical details

Standard HyperSpy signals load the data into memory for fast access and processing. While this behaviour gives good performance in terms of speed, it obviously requires at least as much computer memory as the dataset, and often twice that to store the results of subsequent computations. This can become a significant problem when processing very large datasets on consumer-oriented hardware.

HyperSpy offers a solution for this problem by including `LazySignal` and its derivatives. The main idea of these classes is to perform any operation (as the name suggests) *lazily* (delaying the execution until the result is requested (e.g. saved, plotted)) and in a *blocked* fashion. This is achieved by building a “history tree” (formally called a Directed Acyclic Graph (DAG)) of the computations, where the original data is at the root, and any further operations branch from it. Only when a certain branch result is requested, the way to the root is found and evaluated in the correct sequence on the correct blocks.

The “magic” is performed by (for the sake of simplicity) storing the data not as `numpy.ndarray`, but `dask.array.Array` (see the [dask documentation](#)). `dask` offers a couple of advantages:

- **Arbitrary-sized data processing is possible.** By only loading a couple of chunks at a time, theoretically any signal can be processed, albeit slower. In practice, this may be limited: (i) some operations may require certain chunking pattern, which may still saturate memory; (ii) many chunks should fit into the computer memory comfortably at the same time.
- **Loading only the required data.** If a certain part (chunk) of the data is not required for the final result, it will not be loaded at all, saving time and resources.
- **Able to extend to a distributed computing environment (clusters).** `:dask.distributed` (see the [dask documentation](#)) offers a straightforward way to expand the effective memory for computations to that of a cluster, which allows performing the operations significantly faster than on a single machine.

REGION OF INTEREST (ROI)

ROIs can be defined to select part of any compatible signal and may be applied either to the navigation or to the signal axes. A number of different ROIs are available:

- *Point1DROI*
- *Point2DROI*
- *SpanROI*
- *RectangularROI*
- *CircleROI*
- *Line2DROI*

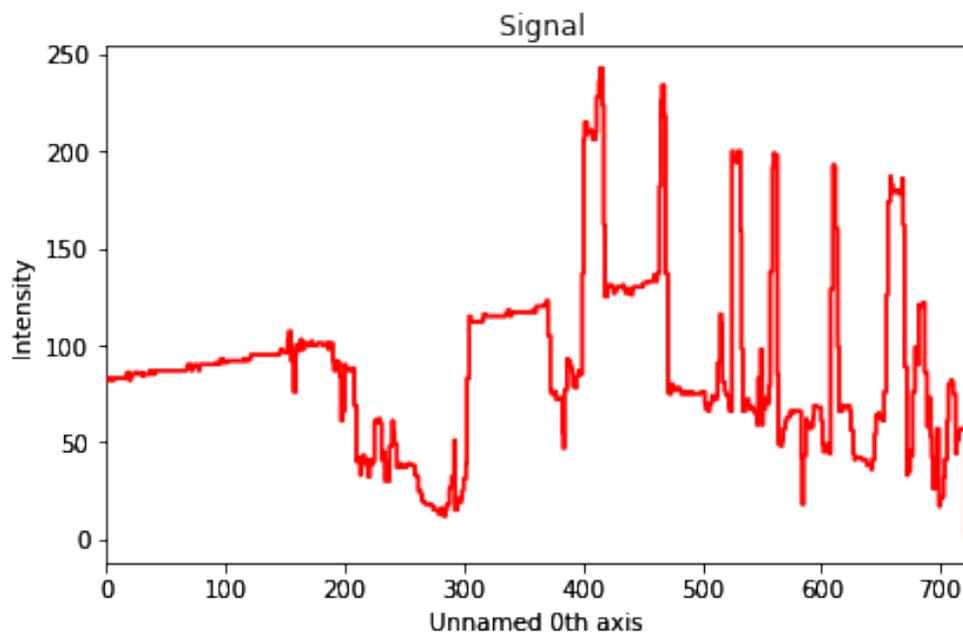
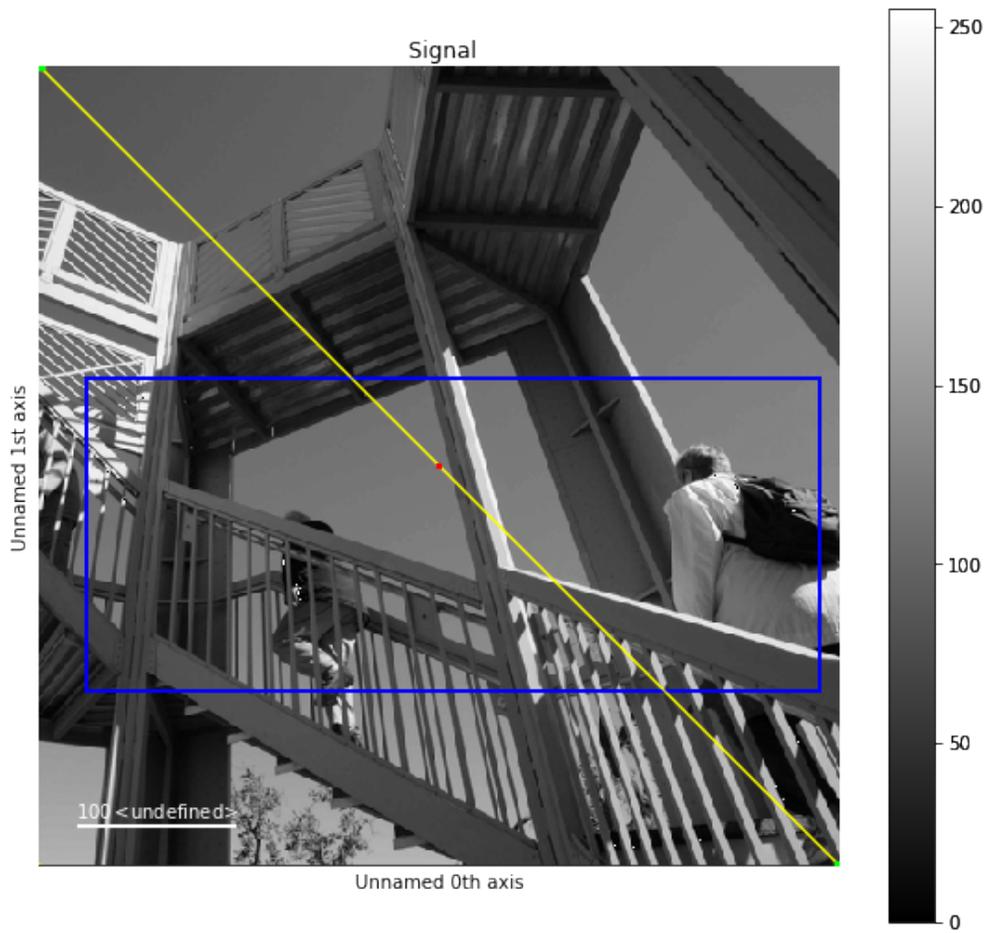
Once created, an ROI can be applied to the signal:

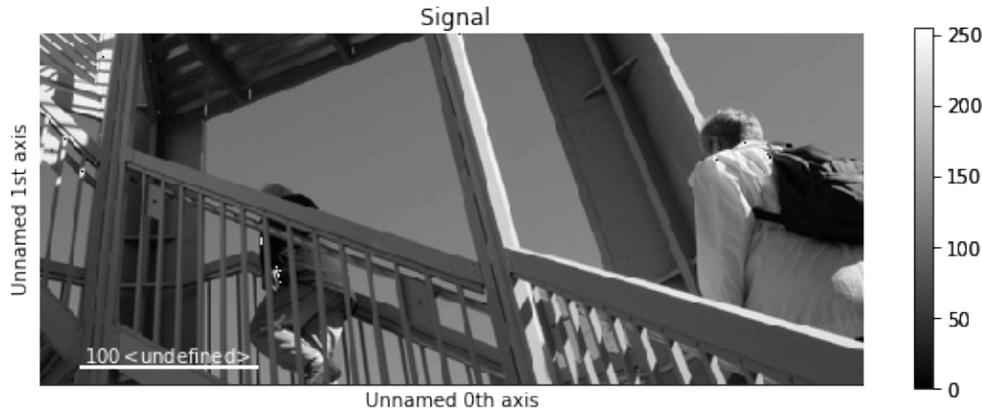
```
>>> s = hs.signals.Signal1D(np.arange(2000).reshape((20,10,10)))
>>> im = hs.signals.Signal2D(np.arange(100).reshape((10,10)))
>>> roi = hs.roi.RectangularROI(left=3, right=7, top=2, bottom=5)
>>> sr = roi(s)
>>> sr
<Signal1D, title: , dimensions: (4, 3|10)>
>>> imr = roi(im)
>>> imr
<Signal2D, title: , dimensions: (|4, 3)>
```

ROIs can also be used *interactively* with widgets. The following example shows how to interactively apply ROIs to an image. Note that *it is necessary* to plot the signal onto which the widgets will be added before calling *interactive()*.

```
>>> import scipy
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> rectangular_roi = hs.roi.RectangularROI(left=30, right=500,
...                                       top=200, bottom=400)
>>> line_roi = hs.roi.Line2DROI(0, 0, 512, 512, 1)
>>> point_roi = hs.roi.Point2DROI(256, 256)
>>> im.plot()
>>> roi2D = rectangular_roi.interactive(im, color="blue")
>>> roi1D = line_roi.interactive(im, color="yellow")
>>> roi0D = point_roi.interactive(im, color="red")
```

Note: Depending on your screen and display settings, it can be difficult to *pick* or manipulate widgets and you can try to change the pick tolerance in the *HyperSpy plot preferences*. Typically, using a 4K resolution with a small scaling





factor (<150 %), setting the pick tolerance to 15 instead of 7.5 makes the widgets easier to manipulate.

If instantiated without arguments, (i.e. `rect = RectangularROI()`) the roi will automatically determine sensible values to center it when interactively adding it to a signal. This provides a convenient starting point to further manipulate the ROI, either by hand or using the gui (i.e. `rect.gui`).

Notably, since ROIs are independent from the signals they sub-select, the widget can be plotted on a different signal altogether.

```
>>> import scipy
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> s = hs.signals.Signal1D(np.random.rand(512, 512, 512))
>>> roi = hs.roi.RectangularROI(left=30, right=77, top=20, bottom=50)
>>> s.plot() # plot signal to have where to display the widget
>>> imr = roi.interactive(im, navigation_signal=s, color="red")
>>> roi(im).plot()
```

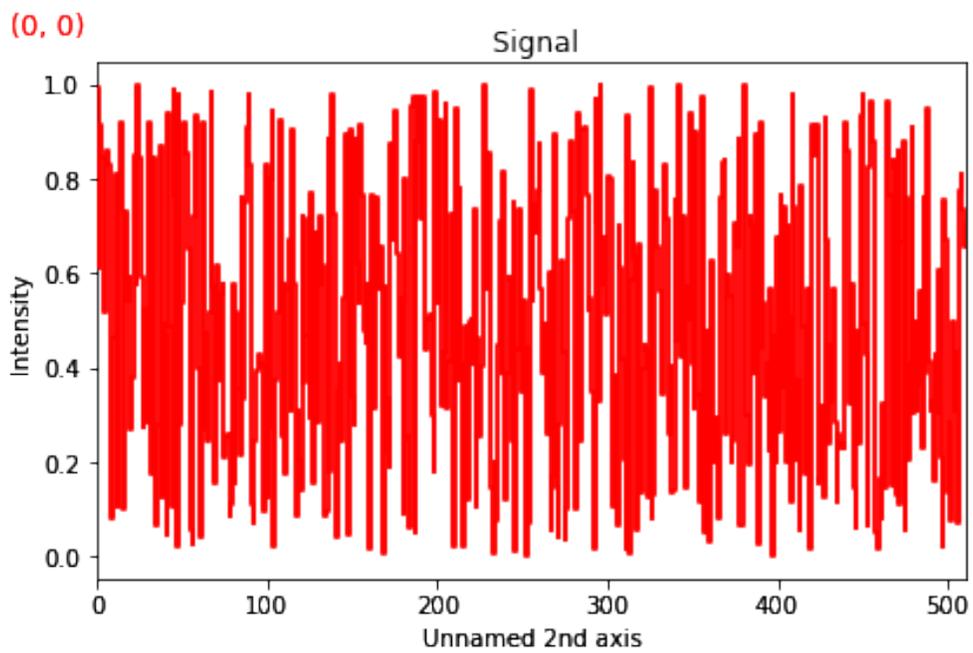
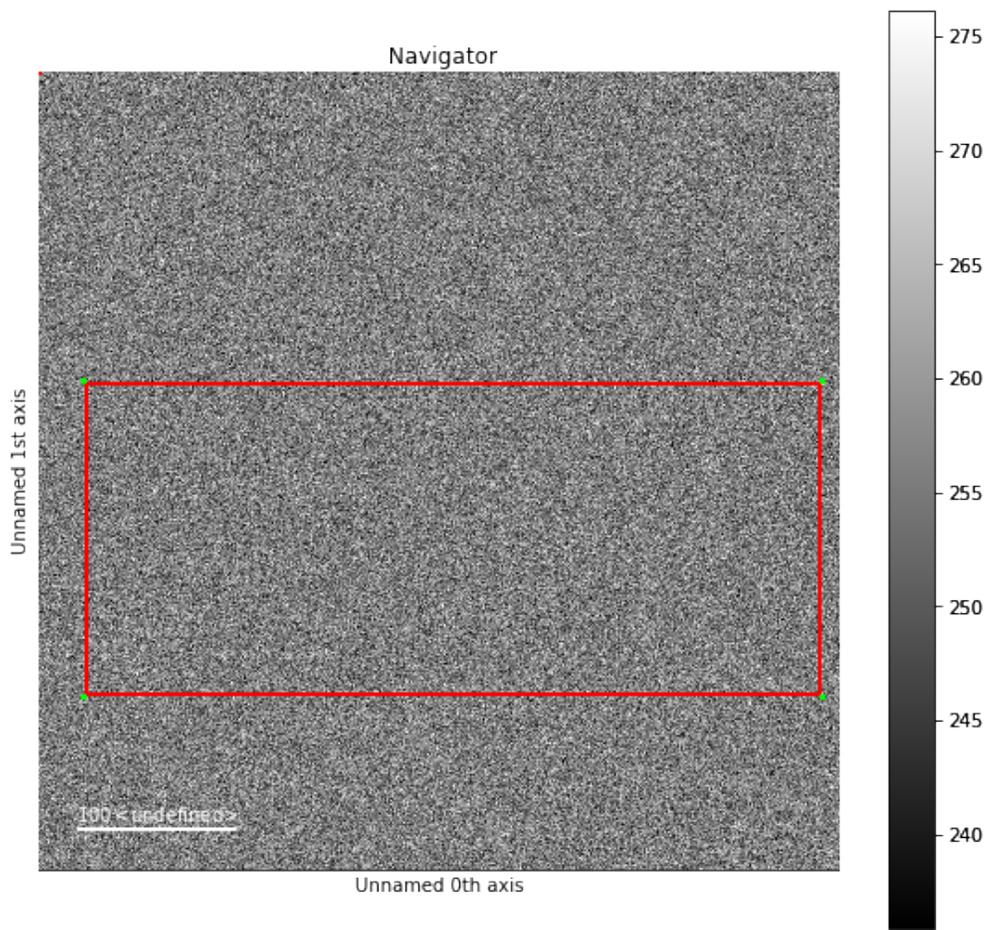
ROIs are implemented in terms of physical coordinates and not pixels, so with proper calibration will always point to the same region.

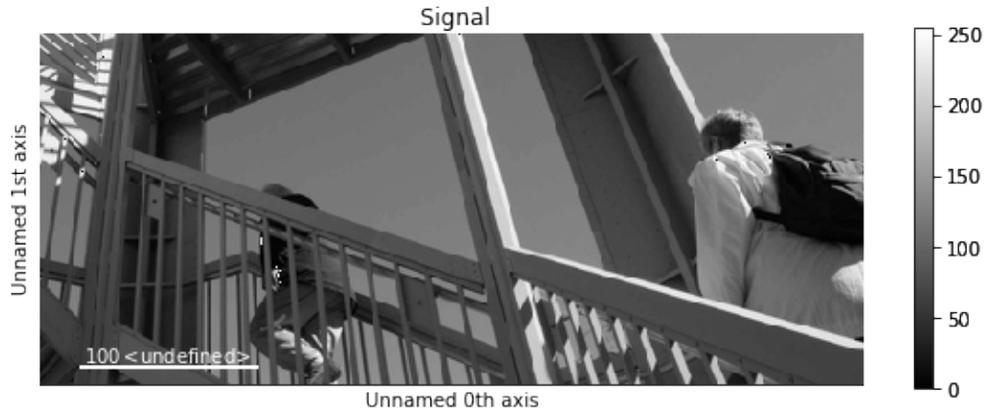
And of course, as all interactive operations, interactive ROIs are chainable. The following example shows how to display interactively the histogram of a rectangular ROI. Notice how we customise the default event connections in order to increase responsiveness.

```
>>> import scipy
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> im.plot()
>>> roi = hs.roi.RectangularROI(left=30, right=500, top=200, bottom=400)
>>> im_roi = roi.interactive(im, color="red")
>>> roi_hist = hs.interactive(im_roi.get_histogram,
...                           event=roi.events.changed,
...                           bins=150, # Set number of bins for `get_histogram`
...                           recompute_out_event=None)
>>> roi_hist.plot()
```

New in version 1.3: ROIs can be used in place of slices when indexing and to define a signal range in functions taken a `signal_range` argument.

All ROIs have a `gui` method that displays an user interface if a hyperspy GUI is installed (currently only works with the `hyperspy_gui_ipywidgets` GUI), enabling precise control of the ROI parameters:





```
>>> # continuing from above:
>>> roi.gui()
```

New in version 1.4: `angle()` can be used to calculate an angle between ROI line and one of the axes providing its name through optional argument `axis`:

```
>>> import scipy
>>> ima = hs.signals.Signal2D(scipy.datasets.ascent())
>>> roi = hs.roi.Line2DROI(x1=144, y1=240, x2=306, y2=178, linewidth=0)
>>> ima.plot()
>>> roi.interactive(ima, color='red')
<BaseSignal, title: , dimensions: (|175)>
```

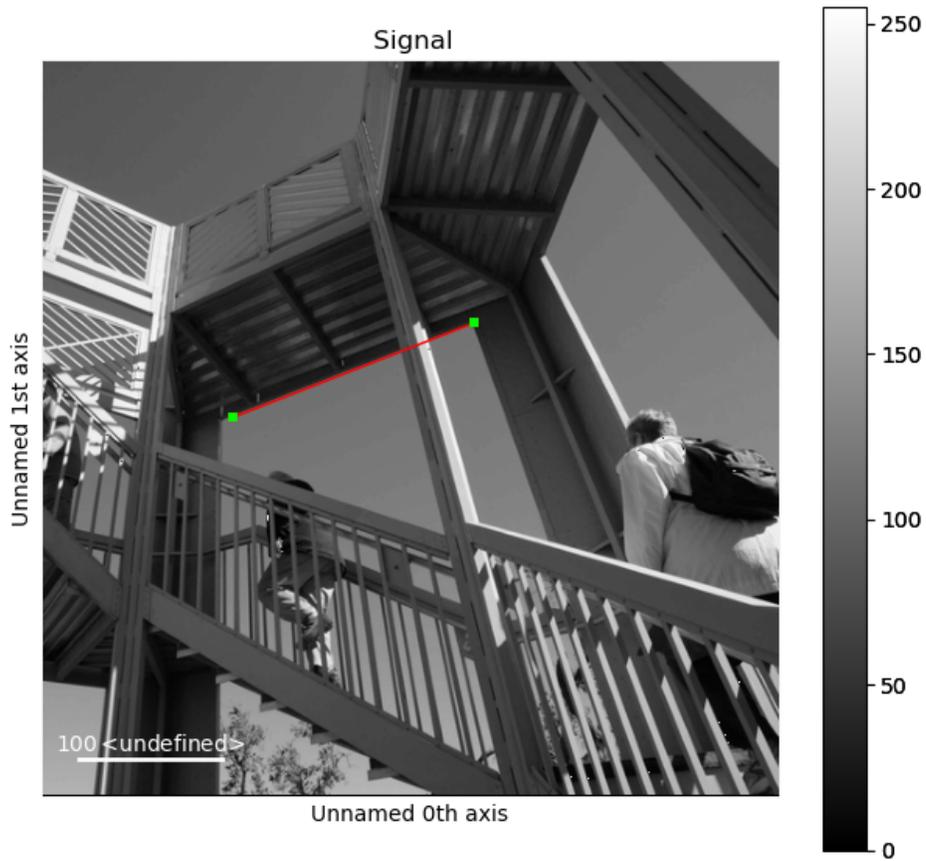
```
>>> roi.angle(axis='vertical')
110.94265054998827
```

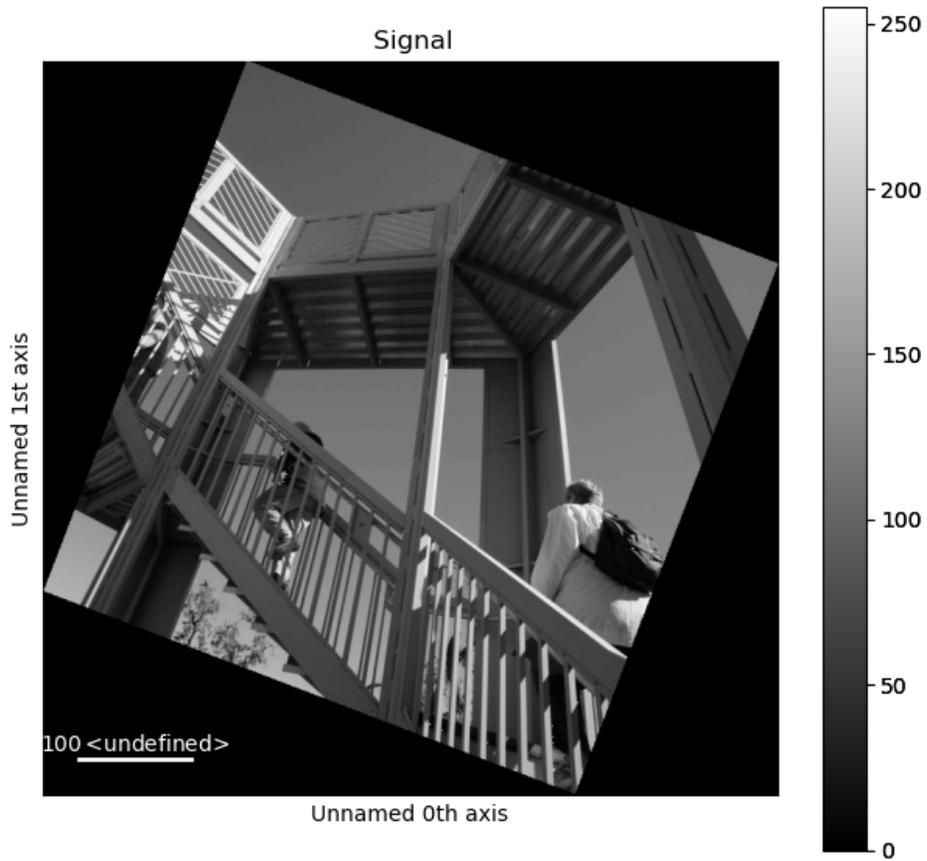
The default output of the method is in degrees, though radians can be selected as follows:

```
>>> roi.angle(axis='vertical', units='radians')
1.9363145329867932
```

Conveniently, `angle()` can be used to rotate an image to align selected features with respect to vertical or horizontal axis:

```
>>> ima.map(scipy.ndimage.rotate, angle=roi.angle(axis='horizontal'), inplace=False).
↳plot()
```





12.1 Slicing using ROIs

ROIs can be used in place of slices when indexing. For example:

```
>>> s = hs.data.two_gaussians()
>>> roi = hs.roi.SpanROI(left=5, right=15)
>>> sc = s.isig[roi]
>>> im = hs.signals.Signal2D(scipy.datasets.ascent())
>>> roi = hs.roi.RectangularROI(left=120, right=460., top=300, bottom=560)
>>> imc = im.isig[roi]
```

New in version 1.3: `gui` method added, for example `gui()`.

New in version 1.6: New `__getitem__` method for all ROIs.

In addition, all ROIs have a `__getitem__` method that enables using them in place of tuples. For example, the method `align2D()` takes a `roi` argument with the left, right, top, bottom coordinates of the ROI. Handily, we can pass a `RectangularROI` ROI instead.

```
>>> import hyperspy.api as hs
>>> import numpy as np
>>> im = hs.signals.Signal2D(np.random.random((10,30,30)))
>>> roi = hs.roi.RectangularROI(left=2, right=10, top=0, bottom=5)
>>> tuple(roi)
(2.0, 10.0, 0.0, 5.0)
>>> im.align2D(roi=roi)
```

12.2 Interactively Slicing Signal Dimensions

`plot_roi_map()` is a function that allows you to interactively visualize the spatial variation of intensity in a Signal within a ROI of its signal axes. In other words, it shows maps of the integrated signal for custom ranges along the signal axis.

To allow selection of the signal ROIs, a plot of the mean signal over all spatial positions is generated. Interactive ROIs can then be adjusted to the desired regions within this plot.

For each ROI, a plot reflecting how the intensity of signal within this ROI varies over the spatial dimensions of the Signal object is also plotted.

For Signal objects with 1 signal dimension `SpanROIs` are used and for 2 signal dimensions, `RectangularROIs` are used.

In the example below, for a hyperspectral map with 2 navigation dimensions and 1 signal dimension (i.e. a spectrum at each position in a 2D map), `SpanROIs` are used to select spectral regions of interest. For each spectral region of interest a plot is generated displaying the intensity within this region at each position in the map.

```
>>> import hyperpsy.api as hs
>>> sig = hs.load('mydata.sur')
>>> sig
<Signal1D, dimensions: (128, 128|1024)>
>>> hs.plot.plot_roi_map(sig, rois=2)
```

Events are a mechanism to send notifications. HyperSpy events are decentralised, meaning that there is not a central events dispatcher. Instead, each object that can emit events has an `events` attribute that is an instance of `Events` and that contains instances of `Event` as attributes. When triggered the first keyword argument, `obj` contains the object that the events belongs to. Different events may be triggered by other keyword arguments too.

13.1 Connecting to events

The following example shows how to connect to the `index_changed` event of `DataAxis` that is triggered with `obj` and `index` keywords:

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> nav_axis = s.axes_manager.navigation_axes[0]
>>> nav_axis.name = "x"
>>> def on_index_changed(obj, index):
...     print("on_index_changed_called")
...     print("Axis name: ", obj.name)
...     print("Index: ", index)

>>> nav_axis.events.index_changed.connect(on_index_changed)
>>> s.axes_manager.indices = (3,)
on_index_changed_called
Axis name: x
Index: 3
>>> s.axes_manager.indices = (9,)
on_index_changed_called
Axis name: x
Index: 9
```

It is possible to select the keyword arguments that are passed to the connected. For example, in the following only the `index` keyword argument is passed to `on_index_changed2` and none to `on_index_changed3`:

```
>>> def on_index_changed2(index):
...     print("on_index_changed2_called")
...     print("Index: ", index)

>>> nav_axis.events.index_changed.connect(on_index_changed2, ["index"])
>>> s.axes_manager.indices = (0,)
on_index_changed_called
Axis name: x
```

(continues on next page)

(continued from previous page)

```

Index: 0
on_index_changed2_called
Index: 0
>>> def on_index_changed3():
...     print("on_index_changed3_called")

>>> nav_axis.events.index_changed.connect(on_index_changed3, [])
>>> s.axes_manager.indices = (1,)
on_index_changed_called
Axis name: x
Index: 1
on_index_changed2_called
Index: 1
on_index_changed3_called

```

It is also possible to map trigger keyword arguments to connected function keyword arguments as follows:

```

>>> def on_index_changed4(arg):
...     print("on_index_changed4_called")
...     print("Index: ", arg)

>>> nav_axis.events.index_changed.connect(on_index_changed4, {"index" : "arg"})
>>> s.axes_manager.indices = (4,)
on_index_changed_called
Axis name: x
Index: 4
on_index_changed2_called
Index: 4
on_index_changed3_called
on_index_changed4_called
Index: 4

```

13.2 Suppressing events

The following example shows how to suppress single callbacks, all callbacks of a given event and all callbacks of all events of an object.

```

>>> with nav_axis.events.index_changed.suppress_callback(on_index_changed2):
...     s.axes_manager.indices = (7,)
on_index_changed_called
Axis name: x
Index: 7
on_index_changed3_called
on_index_changed4_called
Index: 7
>>> with nav_axis.events.index_changed.suppress():
...     s.axes_manager.indices = (6,)

>>> with nav_axis.events.suppress():
...     s.axes_manager.indices = (5,)

```

13.3 Triggering events

Although usually there is no need to trigger events manually, there are cases where it is required. When triggering events manually it is important to pass the right keywords as specified in the event docstring. In the following example we change the *data* attribute of a *BaseSignal* manually and we then trigger the `data_changed` event.

```
>>> s = hs.signals.Signal1D(np.random.random((10, 100)))
>>> s.data[:] = 0
>>> s.events.data_changed.trigger(obj=s)
```


INTERACTIVE OPERATIONS

The function `interactive()` simplifies the definition of operations that are automatically updated when an event is triggered. By default the operation is recomputed when the data or the axes of the original signal is changed.

```
>>> s = hs.signals.Signal1D(np.arange(10.))
>>> ssum = hs.interactive(s.sum, axis=0)
>>> ssum.data
array([45.])
>>> s.data /= 10
>>> s.events.data_changed.trigger(s)
>>> ssum.data
array([4.5])
```

Interactive operations can be performed in a chain.

```
>>> s = hs.signals.Signal1D(np.arange(2 * 3 * 4).reshape((2, 3, 4)))
>>> ssum = hs.interactive(s.sum, axis=0)
>>> ssum_mean = hs.interactive(ssum.mean, axis=0)
>>> ssum_mean.data
array([30., 33., 36., 39.])
>>> s.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],

      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> s.data *= 10
>>> s.events.data_changed.trigger(obj=s)
>>> ssum_mean.data
array([300., 330., 360., 390.] )
```


BIBLIOGRAPHY

[Burdet2013]

P. Burdet, J. Vannod, A. Hessler-Wyser, M. Rappaz, and M. Cantoni, “Three-dimensional chemical analysis of laser-welded NiTi–stainless steel wires using a dual-beam FIB,” *Acta Materialia* 61 (2013): 3090–3098 [<https://doi.org/10.1016/j.actamat.2013.01.069>].

[Candes2011]

E. Candes, X. Li, Y. Ma and J. Wright, “Robust principal component analysis?” *J. ACM* 58(3) (2011): 1–37 [<https://arxiv.org/pdf/0912.3599.pdf>].

[Egerton2011]

R. Egerton, “Electron Energy-Loss Spectroscopy in the Electron Microscope,” Springer-Verlag, 2011 [<https://doi.org/10.1007/978-1-4419-9583-4>].

[Feng2013]

J. Feng, H. Xu and S. Yan, “Online Robust PCA via Stochastic Optimization,” *NIPS 2013*, 2013 [<https://papers.nips.cc/paper/5131-online-robust-pca-via-stochastic-optimization>].

[Herraez]

M. Herráez, D. Burton, M. Lalor, and M. Gdeisat, “Fast two-dimensional phase-unwrapping algorithm based on sorting by reliability following a noncontinuous path,” *Applied Optics*, 41(35) (2002): 7437–7444 [<https://doi.org/10.1364/AO.41.007437>].

[Hyvarinen2000]

A. Hyvarinen and E. Oja, “Independent component analysis: algorithms and applications,” *Neural Networks* 13 (2000): 411–430 [[https://doi.org/10.1016/S0893-6080\(00\)00026-5](https://doi.org/10.1016/S0893-6080(00)00026-5)].

[Keenan2004]

M. Keenan and P. Kotula, “Accounting for Poisson noise in the multivariate analysis of ToF-SIMS spectrum images,” *Surf. Interface Anal* 36(3) (2004): 203–212 [<https://onlinelibrary.wiley.com/doi/10.1002/sia.1657>].

[Nicoletti2013]

O. Nicoletti, F. de la Peña, R. Leary, D. Holland, C. Ducati, and P. Midgley, “Three-dimensional imaging of localized surface plasmon resonances of metal nanoparticles,” *Nature* 502 (2013): 80–84 [<https://doi.org/10.1038/nature12469>].

[Pena2010]

F. de la Peña, M.-H. Berger, J.-F. Hochepeid, F. Dynys, O. Stephan, and M. Walls, “Mapping titanium and tin oxide phases using EELS: An application of independent component analysis,” *Ultramicroscopy* 111 (2010): 169–176 [<https://doi.org/10.1016/j.ultramic.2010.10.001>].

[Zhao2016]

R. Zhao and V. Tan, “Online nonnegative matrix factorization with outliers.” *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2016 [<https://doi.org/10.1109/TSP.2016.2620967>, <https://arxiv.org/pdf/1604.02634.pdf>].

[Zhou2011]

T. Zhou and D. Tao, “GoDec: Randomized Low-rank & Sparse Matrix Decomposition in Noisy Case”, *ICML-11* (2011): 33–40 [https://icml.cc/Conferences/2011/papers/41_icmlpaper.pdf].

[Schaffer2004]

Bernhard Schaffer, Werner Grogger and Gerald Kothleitner. “Automated Spatial Drift Correction for EFTEM Image Series.” *Ultramicroscopy* 102, no. 1 (December 2004): 27–36 [<https://doi.org/10.1016/j.ultramic.2004.08.003>].

[Guizar2008]

Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, “Efficient subpixel image registration algorithms”, *Optics Letters* 33, 156-158 (2008). DOI:10.1364/OL.33.000156 [<https://doi.org/10.1364/OL.33.000156>].

[Satopää2011]

Ville Satopää, Jeannie Albrecht, David Irwin, Barath Raghavan. “Finding a “Kneedle” in a Haystack: Detecting Knee Points in System Behavior. 31st International Conference on Distributed Computing Systems Workshops”, pp. 166-171, Minneapolis, Minnesota, USA, June 2011 [<https://doi.org/10.1109/ICDCSW.2011.20>].

[Lerotic2004]

M Lerotic, C Jacobsen, T Schafer, S Vogt “Cluster analysis of soft X-ray spectromicroscopy data”. *Ultramicroscopy* 100 (2004) 35–57 [<https://doi.org/10.1016/j.ultramic.2004.01.008>]

[Iakoubovskii2008]

Iakoubovskii, K., K. Mitsuishi, Y. Nakayama, and K. Furuya. ‘Thickness Measurements with Electron Energy Loss Spectroscopy’. *Microscopy Research and Technique* 71, no. 8 (2008): 626–31. [<https://onlinelibrary.wiley.com/doi/10.1002/jemt.20597>].

[Rossouw2015]

D. Rossouw, P. Burdet, F. de la Peña, C. Ducati, B. Knappett, A. Wheatley, and P. Midgley, “Multicomponent Signal Unmixing from Nanoheterostructures: Overcoming the Traditional Challenges of Nanoscale X-ray Analysis via Machine Learning,” *Nano Lett.* 15(4) (2015): 2716–2720 [<https://doi.org/10.1021/acs.nanolett.5b00449>].

[White2009]

T.A. White, “Structure solution using precession electron diffraction and diffraction tomography” PhD Thesis, University of Cambridge, 2009.

[Zaefferer2000]

S. Zaefferer, “New developments of computer-aided crystallographic analysis in transmission electron microscopy” *J. Appl. Crystallogr.*, vol. 33, no. v, pp. 10–25, 2000. [<https://doi.org/10.1107/S0021889899010894>].

GALLERY OF EXAMPLES

This gallery contains the commented code for short examples illustrating simple tasks that can be performed with HyperSpy.

MARKERS

Gallery of examples on using HyperSpy markers.

SIGNAL CREATION

Below is a gallery of examples on creating a signal and plotting.

LOADING, SAVING AND EXPORTING

Below is a gallery of examples on loading, saving and exporting data.

MODEL FITTING

Below is a gallery of examples on model fitting.

REGION OF INTEREST

Below is a gallery of examples on using regions of interest with HyperSpy signals.

SIMPLE SIMULATIONS

Below is a gallery of examples on simulating signals which can be used to test HyperSpy functionalities

22.1 Markers

Gallery of examples on using HyperSpy markers.

22.1.1 Ragged Points

As for ragged signals, the number of markers at each position can vary and this is done by passing a ragged array to the constructor of the markers.

Create a signal

```
import hyperspy.api as hs
import numpy as np

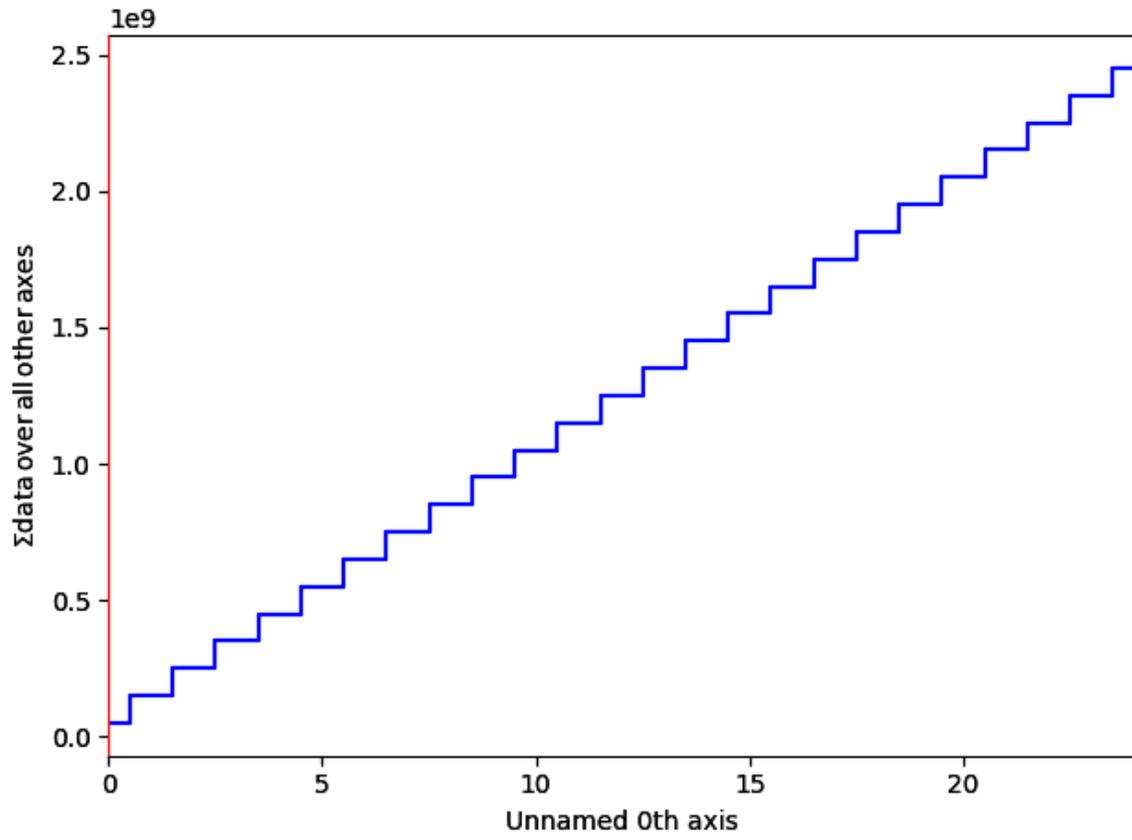
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.arange(25*100*100).reshape((25, 100, 100))
s = hs.signals.Signal2D(data)
```

Create the ragged array with varying number of markers for each navigation position

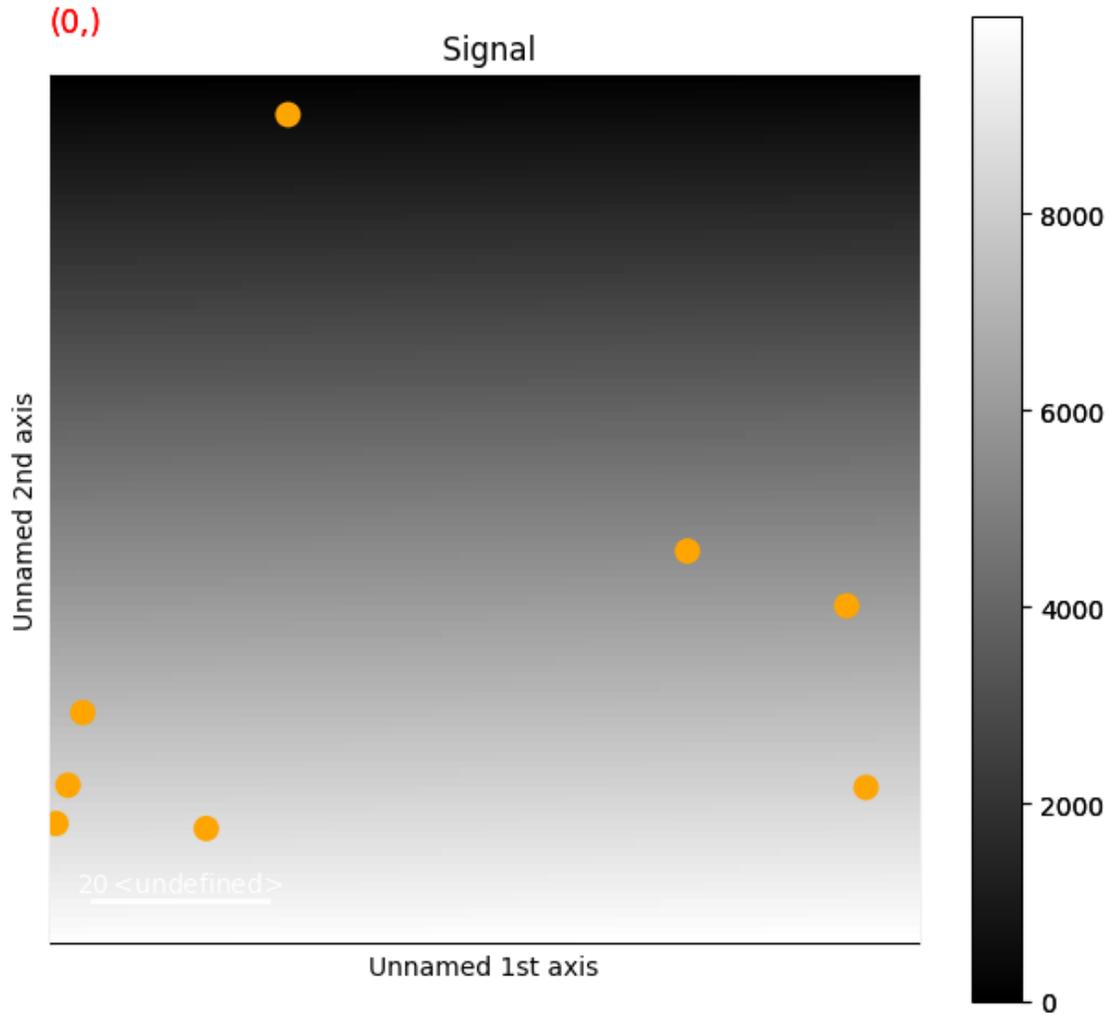
```
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(offsets.shape):
    num = rng.integers(3, 10)
    offsets[ind] = rng.random((num, 2)) * 100

m = hs.plot.markers.Points(
    offsets=offsets,
    facecolor='orange',
)

s.plot()
s.add_marker(m)
```



.



sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 0.856 seconds)

22.1.2 Arrow markers

Create a signal

```
import hyperspy.api as hs
import numpy as np

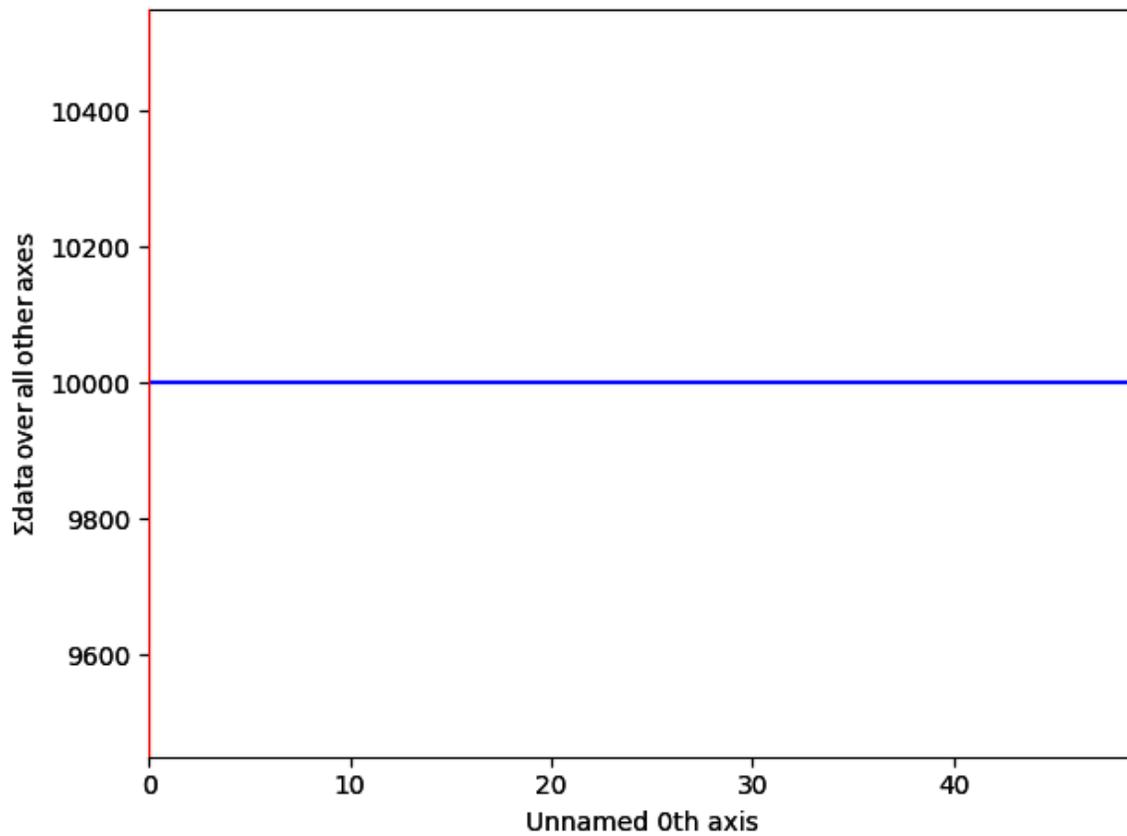
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((50, 100, 100))
s = hs.signals.Signal2D(data)

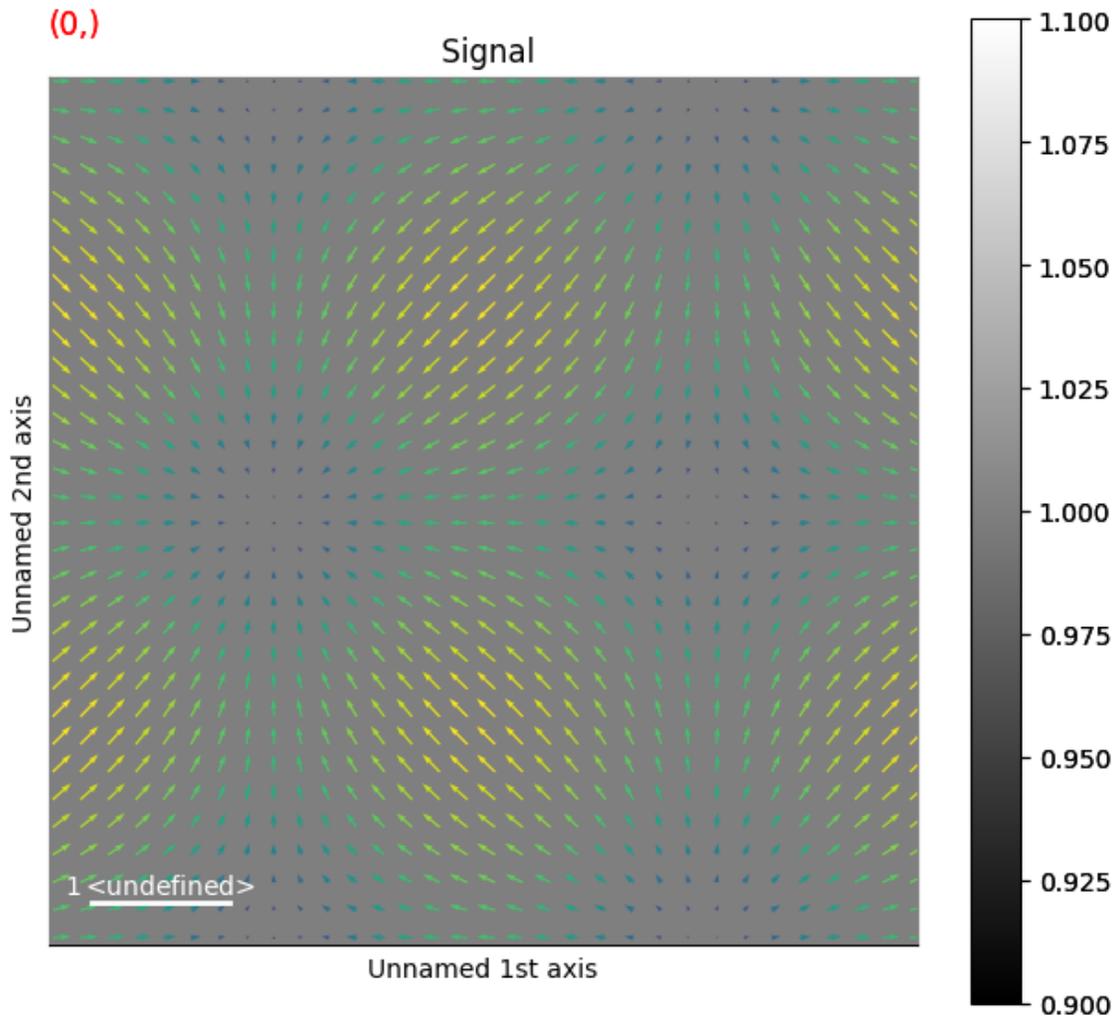
for axis in s.axes_manager.signal_axes:
    axis.scale = 2*np.pi / 100
```

This example shows how to draw arrows

```
# Define the position of the arrows
X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
offsets = np.column_stack((X.ravel(), Y.ravel()))
U = np.cos(X).ravel() / 7.5
V = np.sin(Y).ravel() / 7.5
C = np.hypot(U, V)

m = hs.plot.markers.Arrows(offsets, U, V, C=C)
s.plot()
s.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 0.890 seconds)

22.1.3 Vertical Line Markers

Create a signal

```
import hyperspy.api as hs
import matplotlib.pyplot as plt
import numpy as np

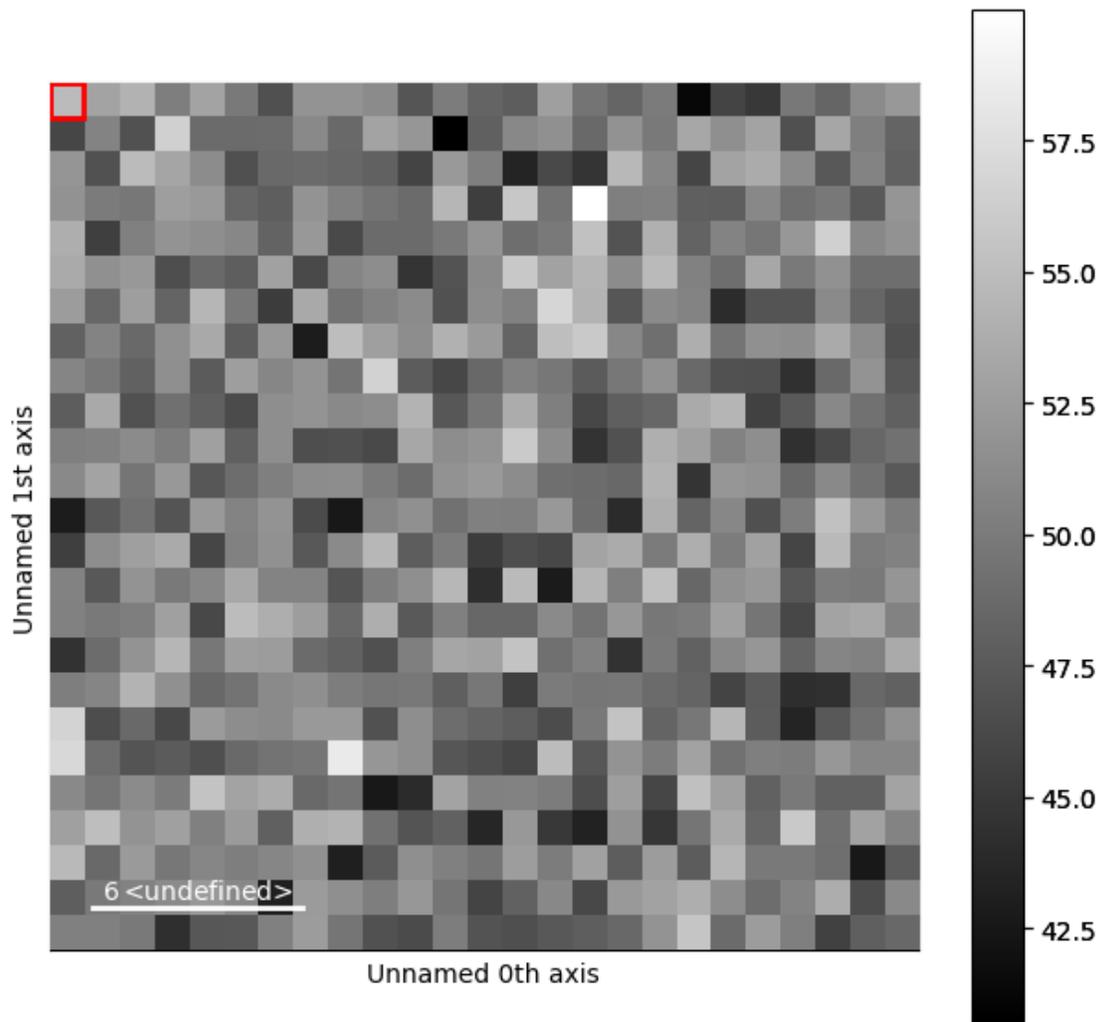
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = rng.random((25, 25, 100))
s = hs.signals.Signal1D(data)
```

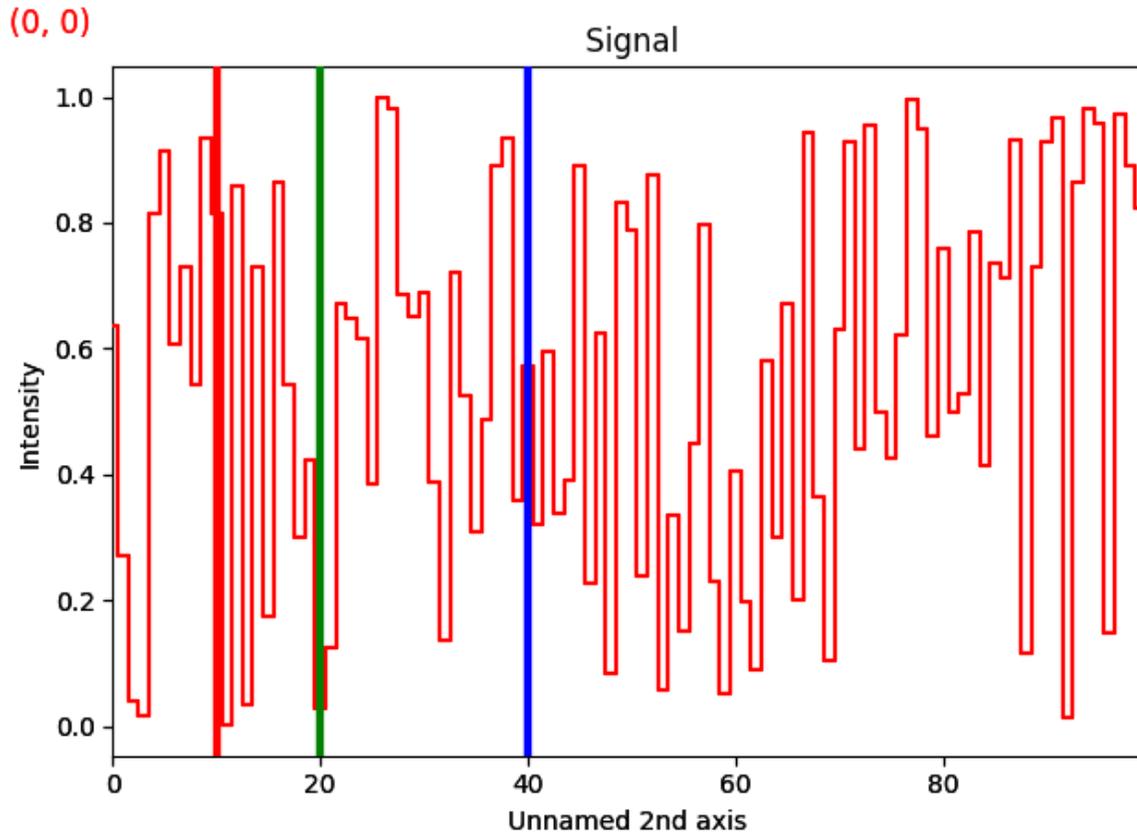
This first example shows how to draw 3 static (same position for all navigation coordinate) vertical lines

```
offsets = np.array([10, 20, 40])

m = hs.plot.markers.VerticalLines(
    offsets=offsets,
    linewidth=3,
    colors=['r', 'g', 'b'],
)

s.plot()
s.add_marker(m)
```





Dynamic Line Markers

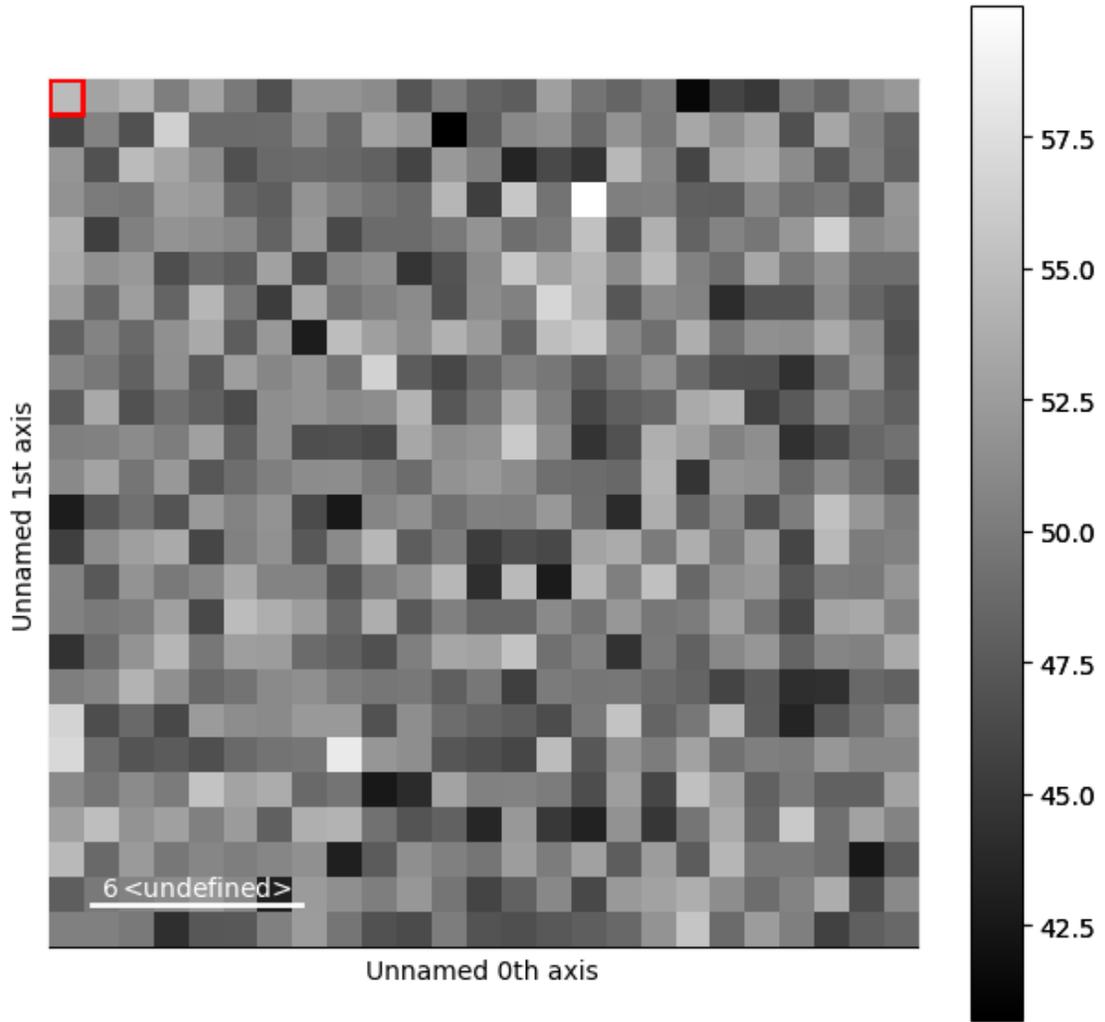
This example shows how to draw dynamic lines markers, whose positions and numbers depends on the navigation coordinates

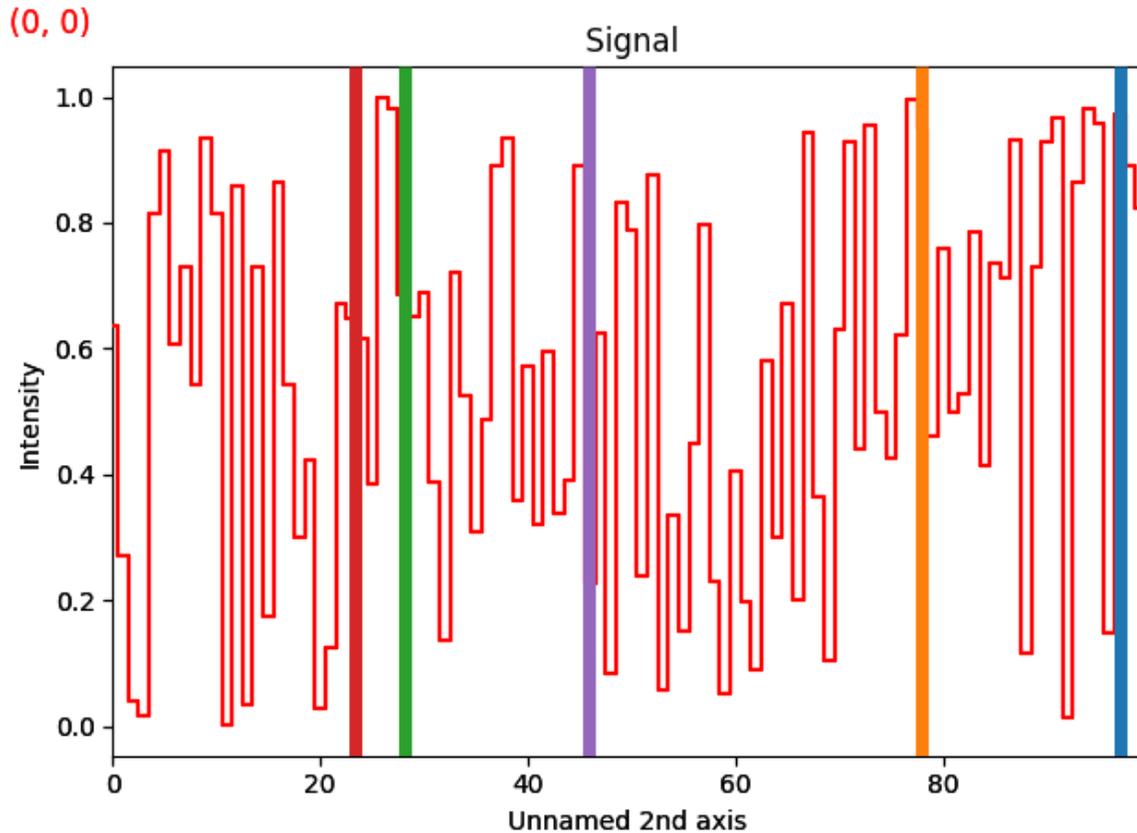
```
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(offsets.shape):
    offsets[ind] = rng.random(rng.integers(10)) * 100

# Get list of colors
colors = list(plt.rcParams['axes.prop_cycle'].by_key()['color'])

m = hs.plot.markers.VerticalLines(
    offsets=offsets,
    linewidth=5,
    colors=colors,
)

s.plot()
s.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.395 seconds)

22.1.4 Circle Markers with Radius Dependent Coloring

This example shows how to draw circle with the color of the circle scaling with the radius of the circle

Create a signal

```
import hyperspy.api as hs
import matplotlib.pyplot as plt
import numpy as np

# Create a Signal2D
rng = np.random.default_rng(0)
s = hs.signals.Signal2D(np.ones((25, 100, 100)))
```

This first example shows how to draw arrows

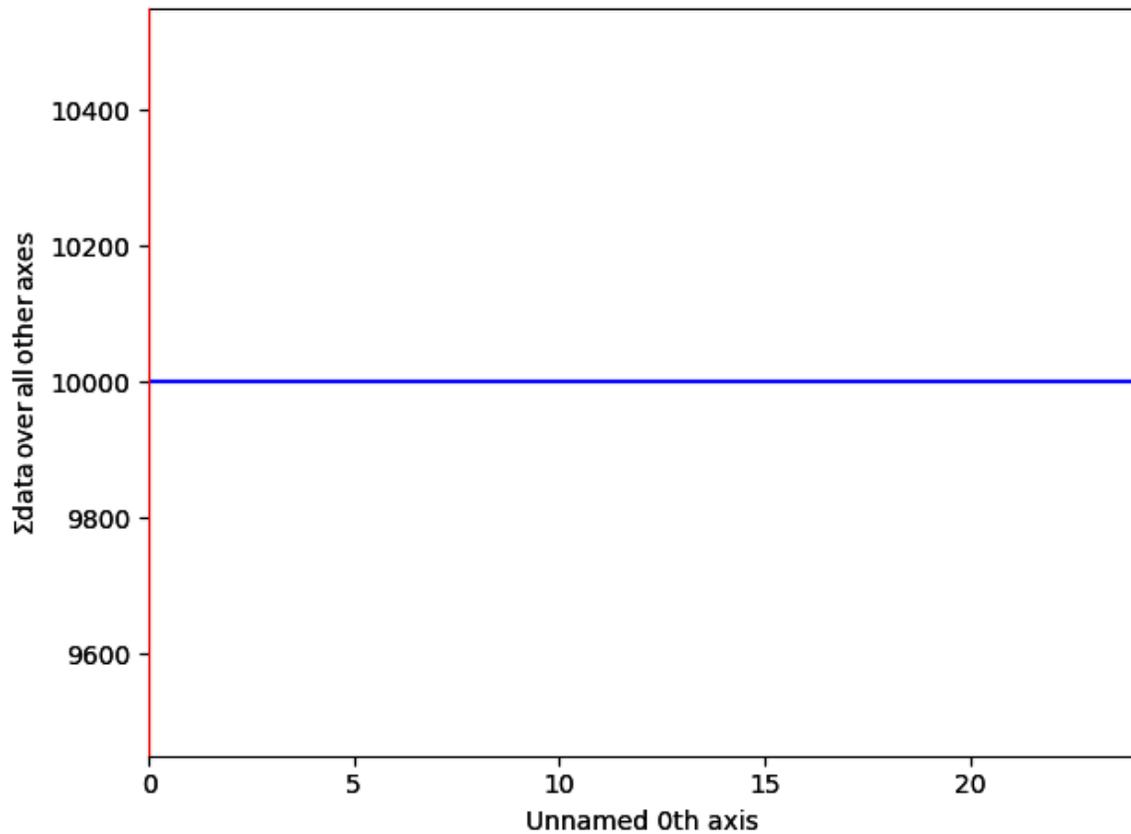
```
# Define the size of the circles
sizes = rng.random((10, )) * 20 + 5

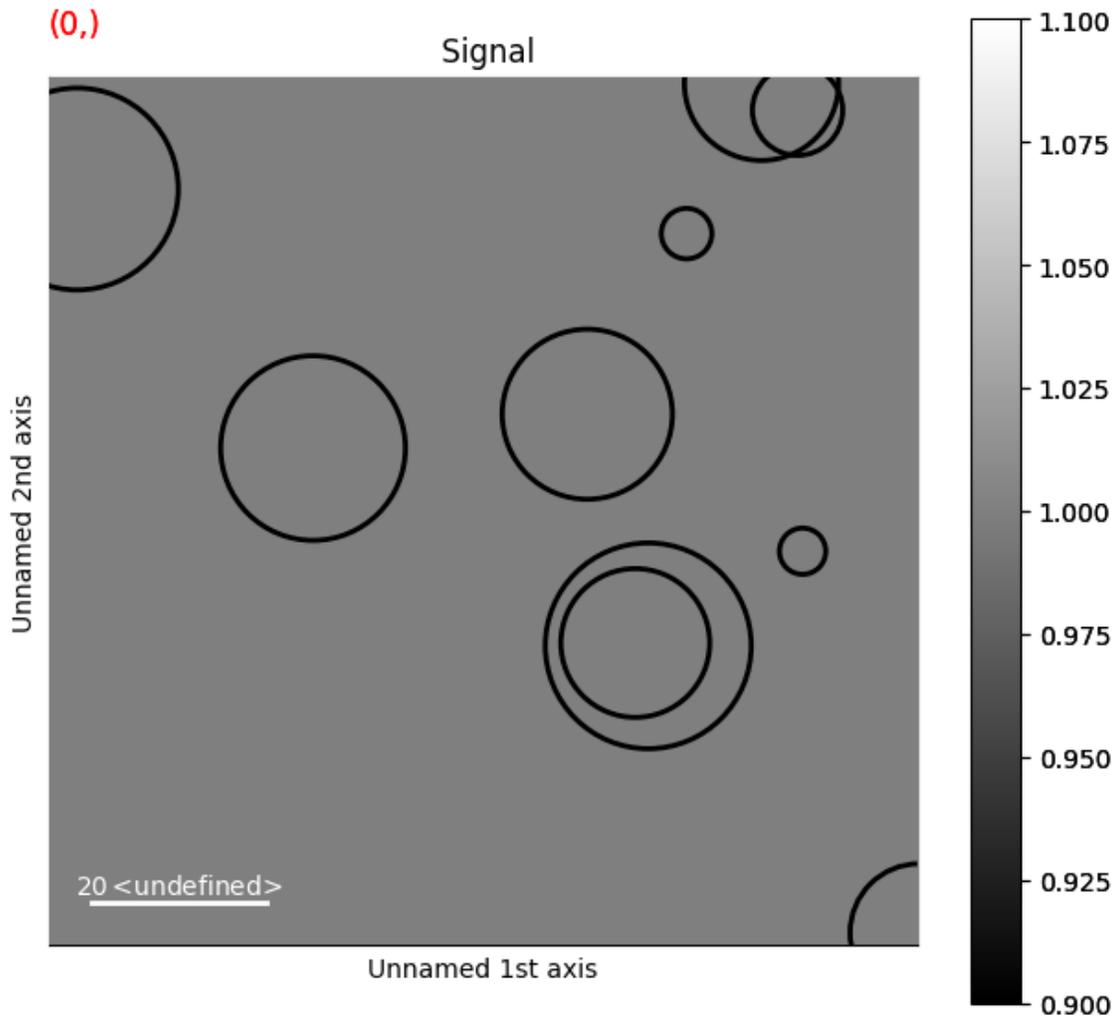
# Define the position of the circles
offsets = rng.random((10, 2)) * 100
```

(continues on next page)

(continued from previous page)

```
m = hs.plot.markers.Circles(  
    sizes=sizes,  
    offsets=offsets,  
    linewidth=2,  
    )  
  
s.plot()  
s.add_marker(m)
```





Note: Any changes to the marker made by setting `matplotlib.collections.Collection` attributes will not be saved when saving as `hspy/zspy` file.

```
# Set the color of the circles
m.set_ScalarMappable_array(sizes.ravel() / 2)

# Add corresponding colorbar
cbar = m.plot_colorbar()
cbar.set_label('Circle radius')

# Set animated state of colorbar to support blitting
animated = plt.gcf().canvas.supports_blit
cbar.ax.yaxis.set_animated(animated)
cbar.solids.set_animated(animated)
```

sphinx_gallery_thumbnail_number =

Total running time of the script: (0 minutes 0.681 seconds)

22.1.5 Text Markers

Create a signal

```
import hyperspy.api as hs
import numpy as np

# Create a Signal2D with 1 navigation dimension
rng = np.random.default_rng(0)
data = np.ones((10, 100, 100))
s = hs.signals.Signal2D(data)
```

This first example shows how to draw static Text markers

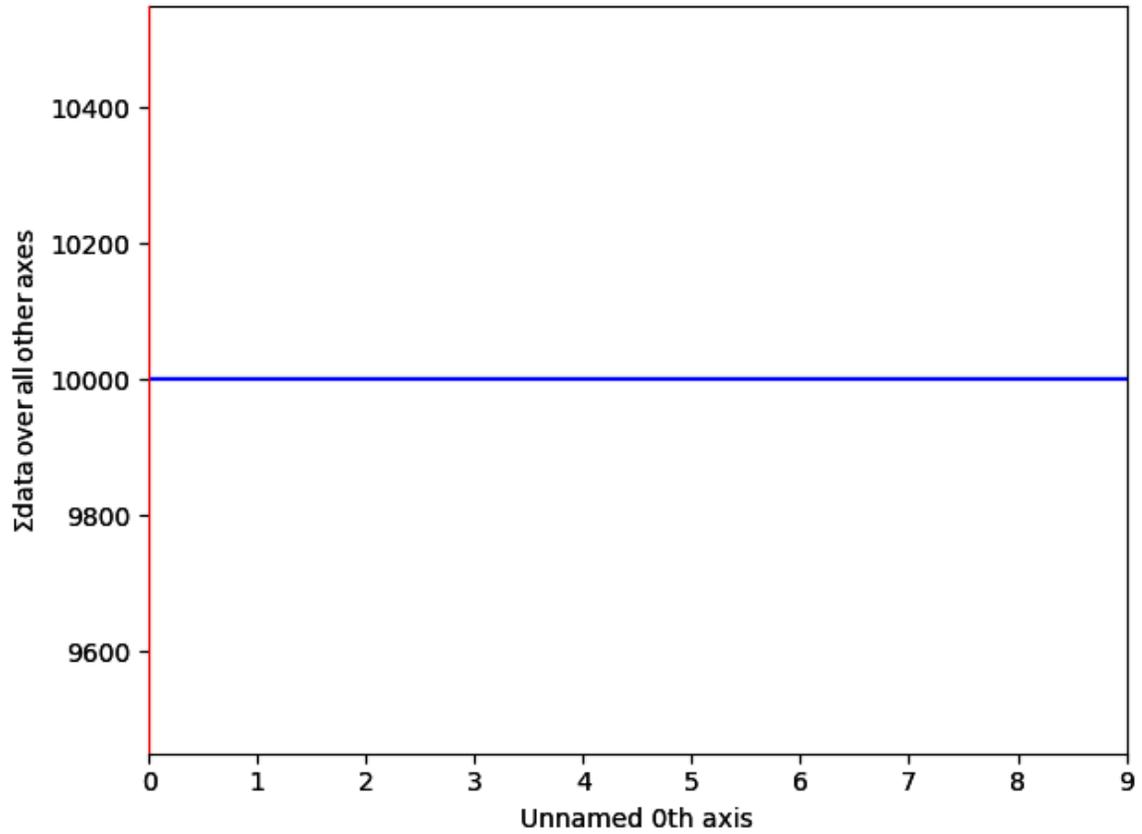
```
# Define the position of the texts
offsets = np.stack([np.arange(0, 100, 10)]*2).T + np.array([5,]*2)
texts = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'f', 'h', 'i'])

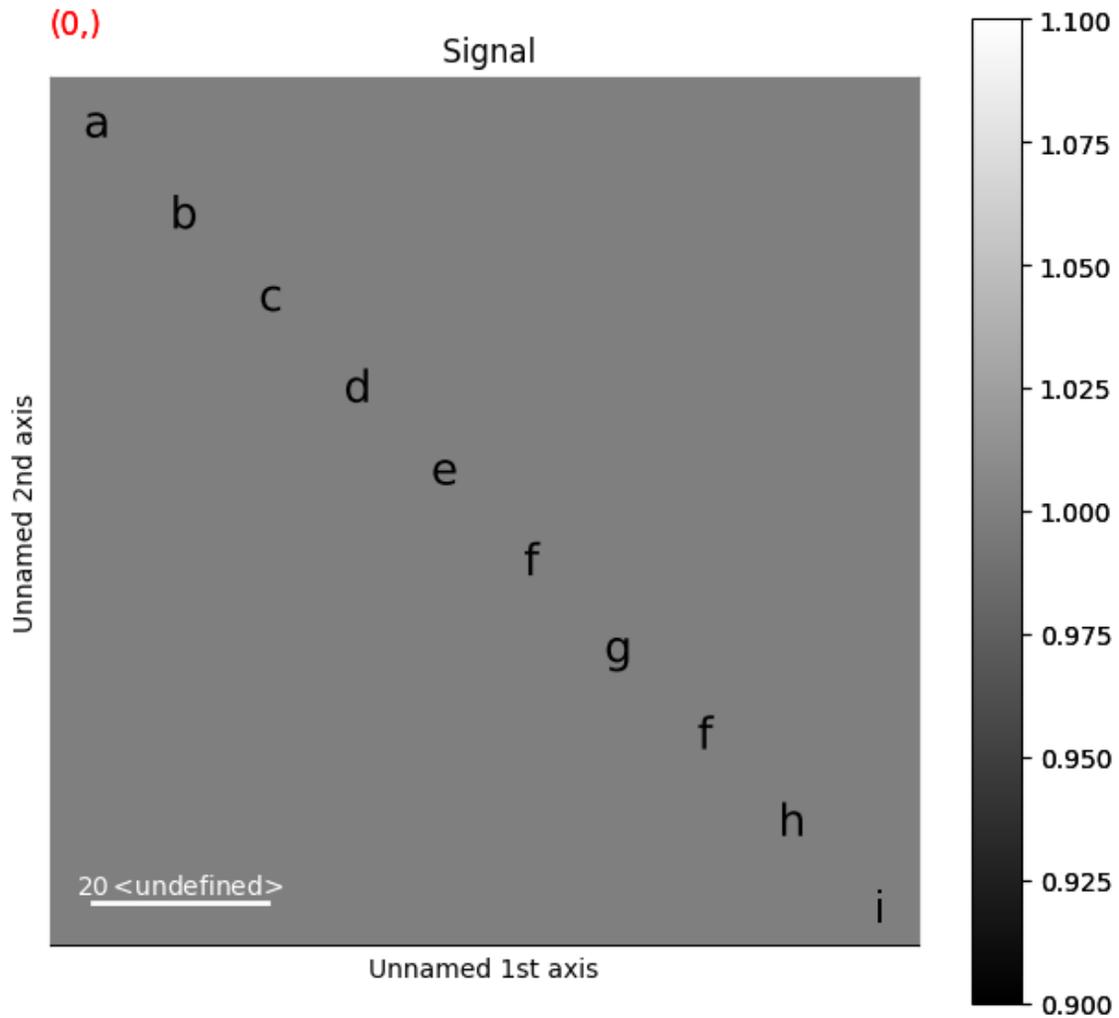
m = hs.plot.markers.Texts(
    offsets=offsets,
```

(continues on next page)

(continued from previous page)

```
texts=texts,  
sizes=3,  
facecolor="black",  
)  
s.plot()  
s.add_marker(m)
```





Dynamic Text Markers

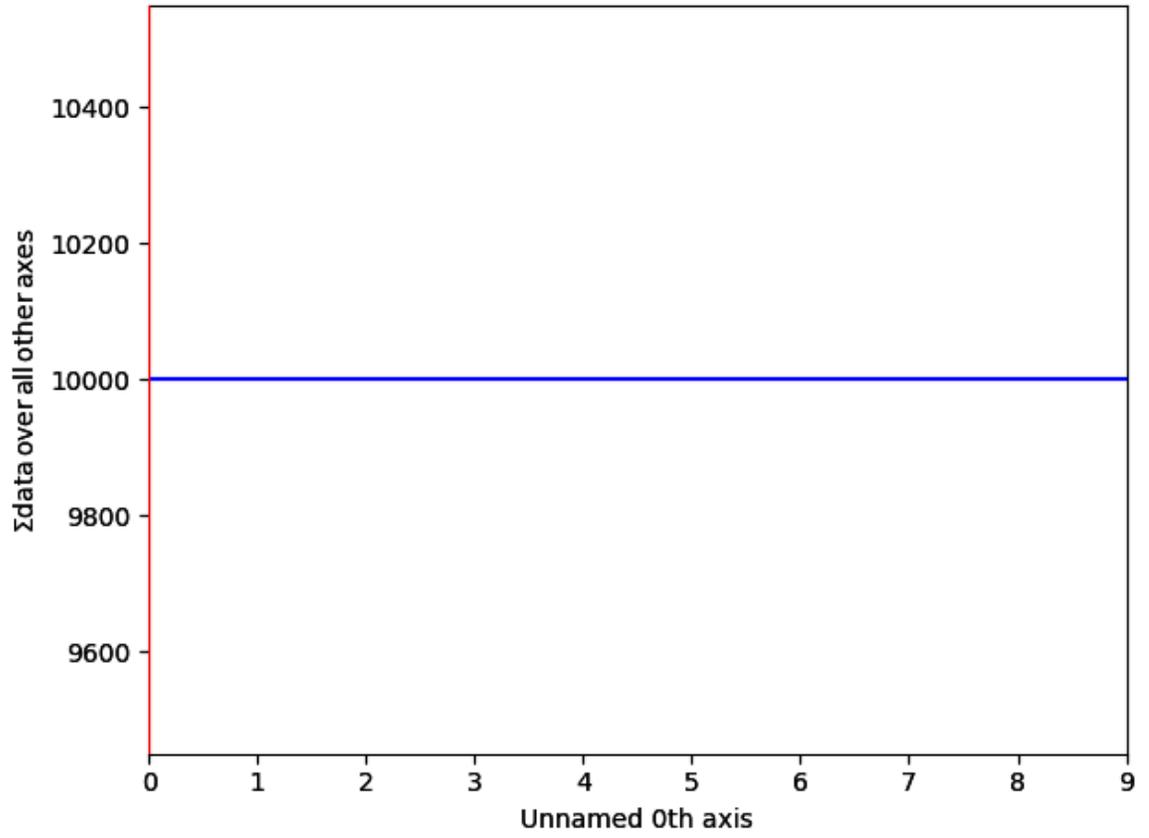
```
s2 = hs.signals.Signal2D(data)

offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)

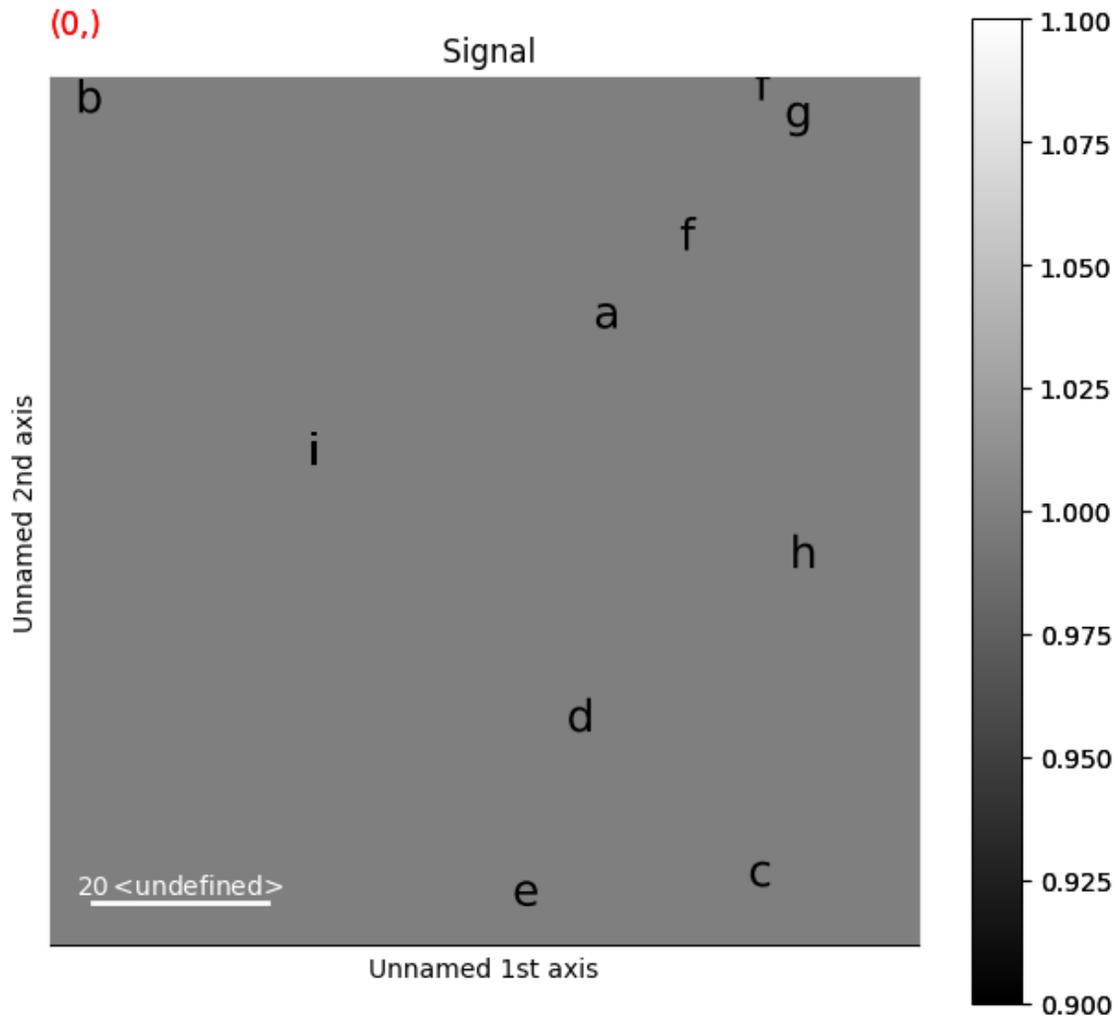
for index in np.ndindex(offsets.shape):
    offsets[index] = rng.random((10, 2)) * 100

m2 = hs.plot.markers.Texts(
    offsets=offsets,
    texts=texts,
    sizes=3,
    facecolor="black",
)

s2.plot()
s2.add_marker(m2)
```



.



sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.367 seconds)

22.1.6 Line Markers

Create a signal

```
import hyperspy.api as hs
import matplotlib.pyplot as plt
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

This first example shows how to draw static stars markers using the matplotlib StarPolygonCollection

```

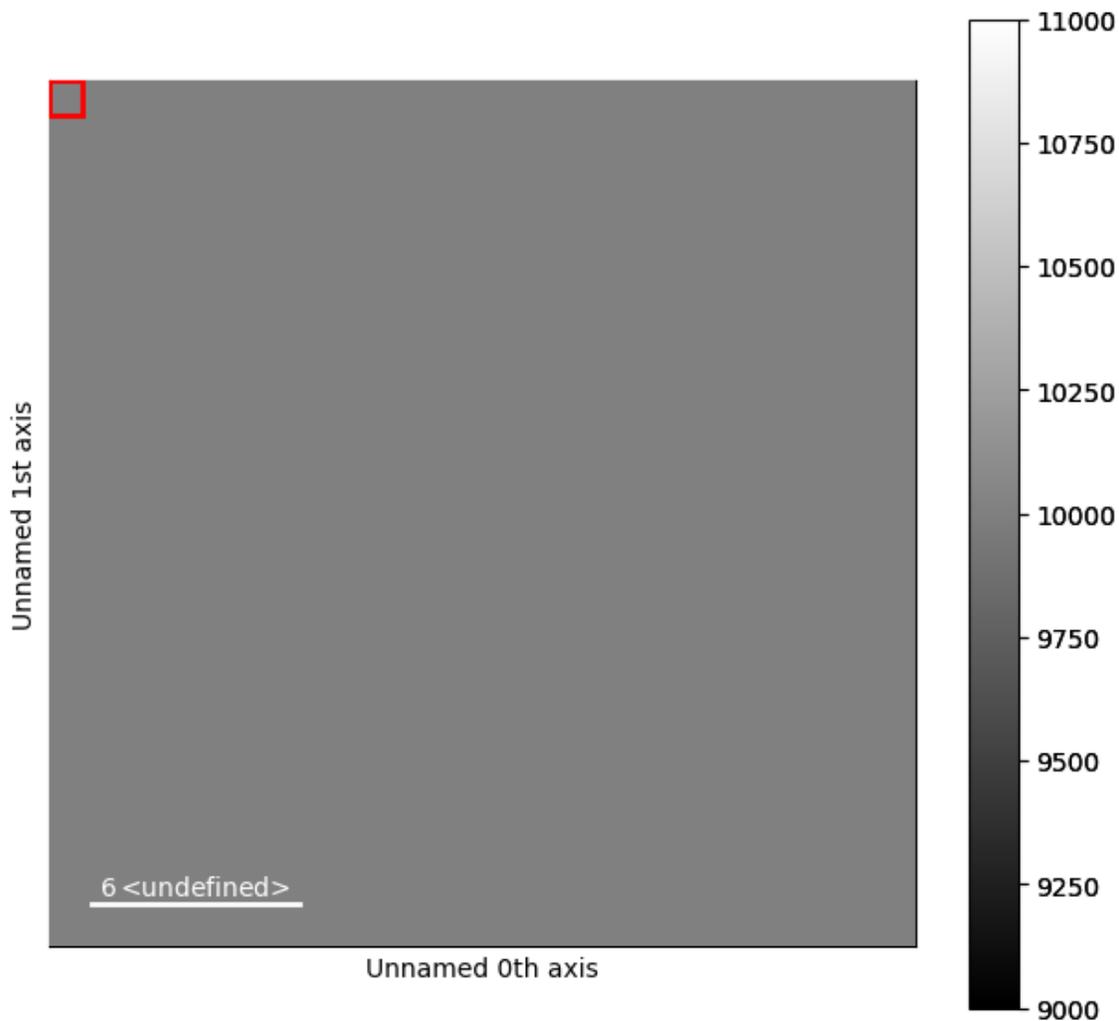
# Define the position of the lines
# line0: (x0, y0), (x1, y1)
# line1: (x0, y0), (x1, y1)
# ...

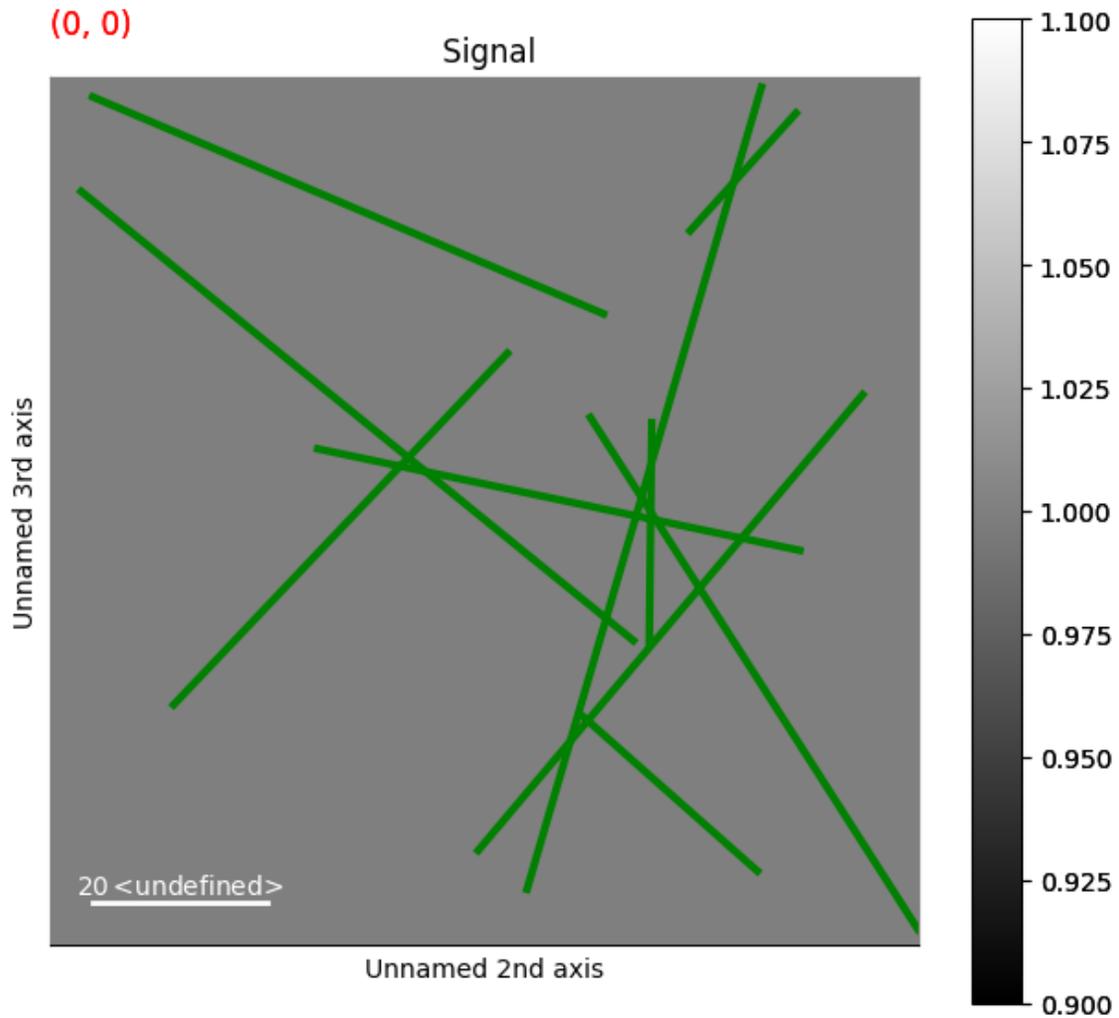
segments = rng.random((10, 2, 2)) * 100

m = hs.plot.markers.Lines(
    segments=segments,
    linewidth=3,
    colors='g',
)

s.plot()
s.add_marker(m)

```





Dynamic Line Markers

This first example shows how to draw dynamic lines markers, whose position depends on the navigation coordinates

```
segments = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(segments.shape):
    segments[ind] = rng.random((10, 2, 2)) * 100

# Get list of colors
colors = list(plt.rcParams['axes.prop_cycle'].by_key()['color'])

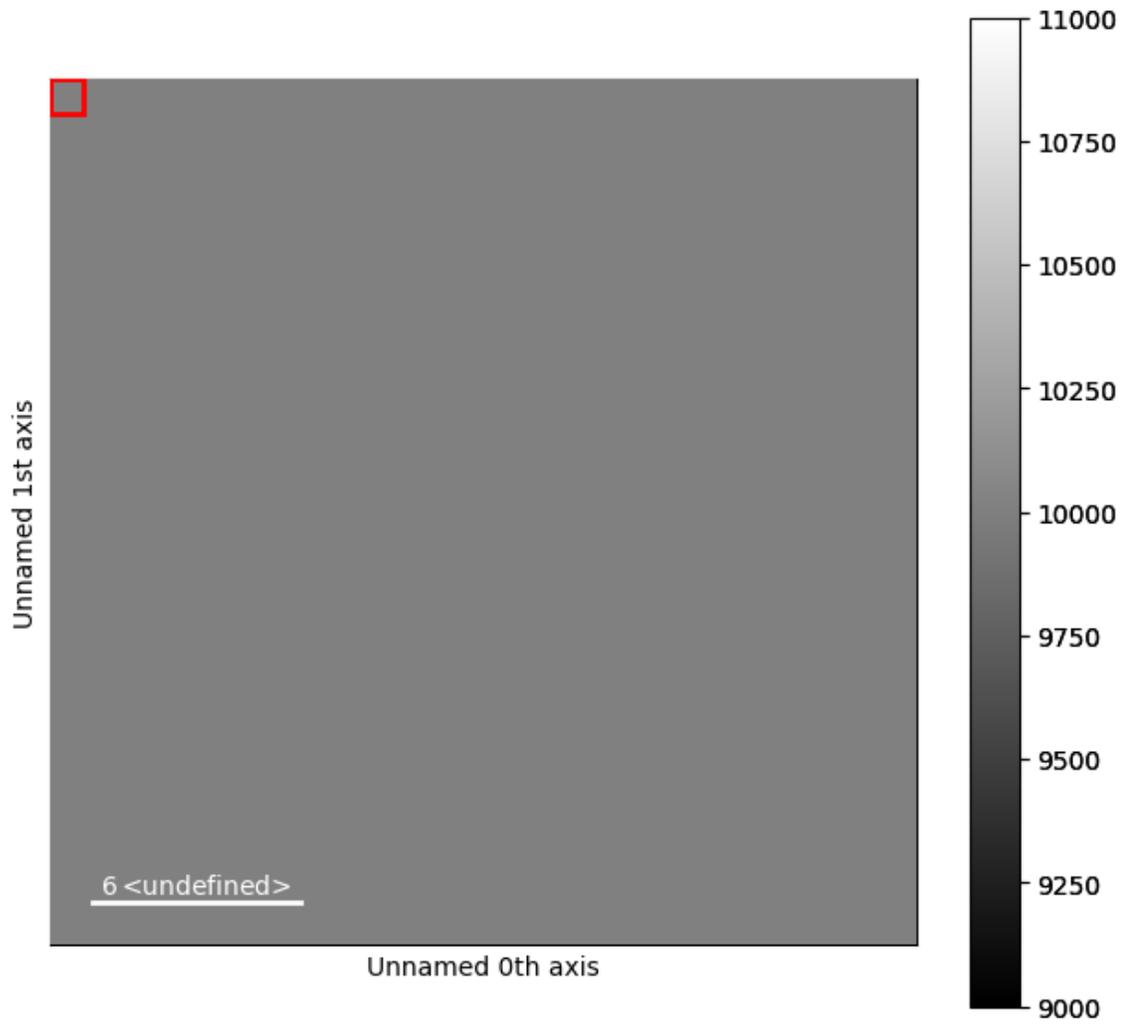
m = hs.plot.markers.Lines(
    segments=segments,
    colors=colors,
    linewidth=5,
)

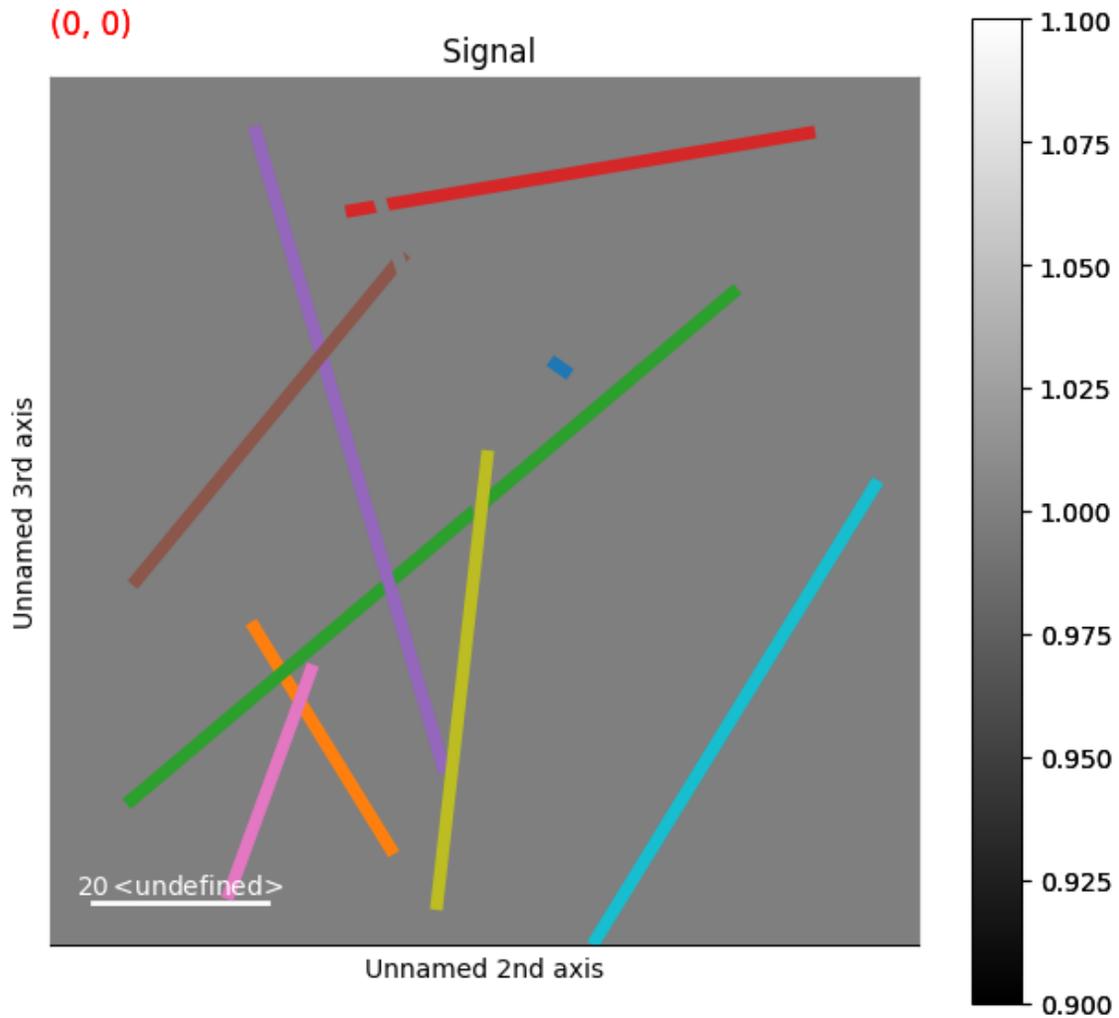
s.plot()
```

(continues on next page)

(continued from previous page)

```
s.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.555 seconds)

22.1.7 Varying number of arrows per navigation position

Create a signal

```
import hyperspy.api as hs
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((10, 100, 100))
s = hs.signals.Signal2D(data)

for axis in s.axes_manager.signal_axes:
    axis.scale = 2*np.pi / 100
```

(continues on next page)

(continued from previous page)

```
# Select navigation position 5
s.axes_manager.indices = (5, )
```

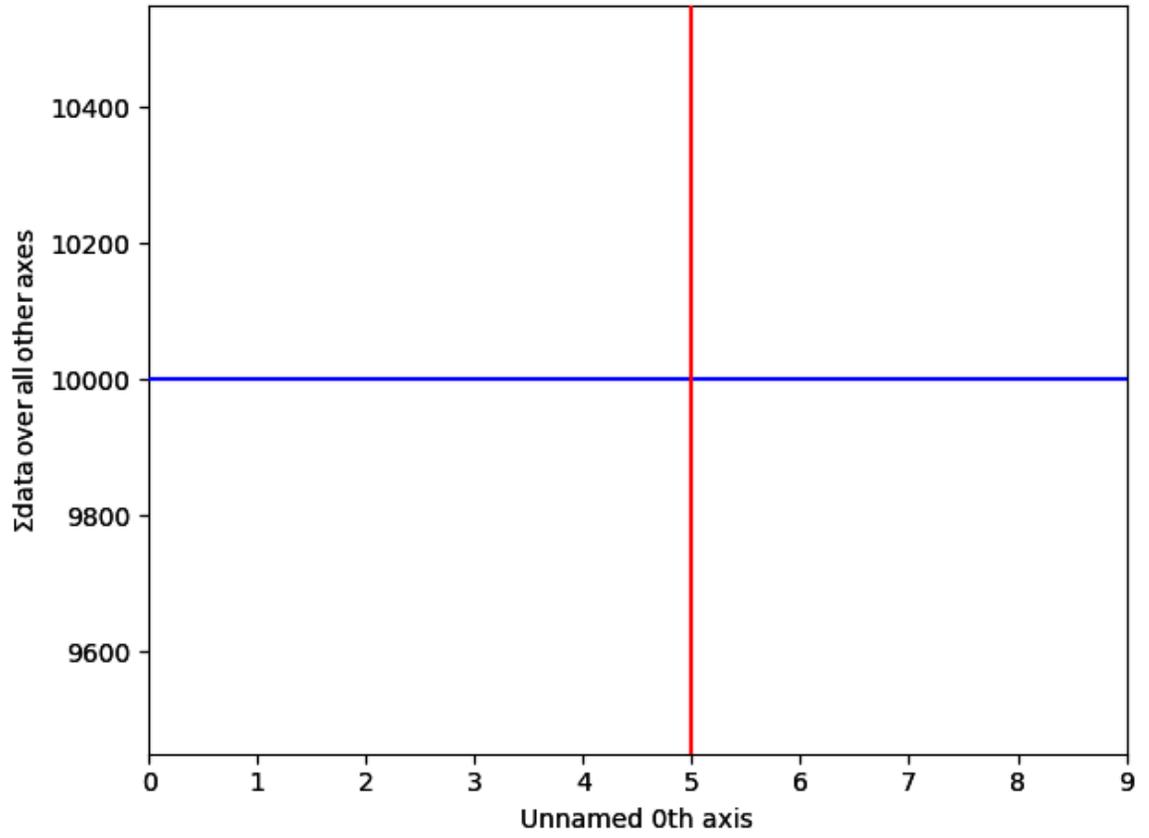
Dynamic Arrow Markers: Changing Length

This example shows how to use the Arrows marker with a varying number of arrows per navigation position

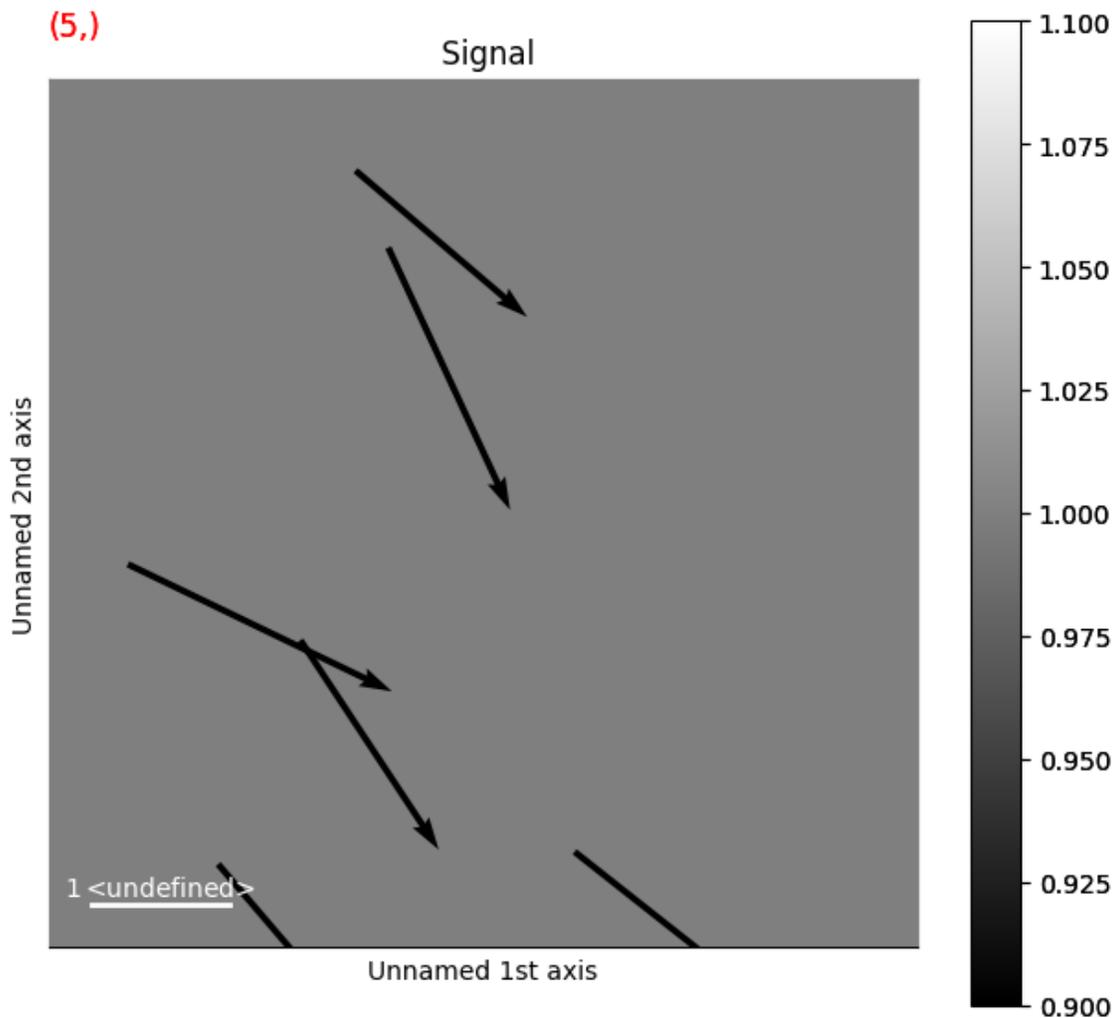
```
# Define the position of the arrows, use ragged array to enable the navigation
# position dependence
offsets= np.empty(s.axes_manager.navigation_shape, dtype=object)
U = np.empty(s.axes_manager.navigation_shape, dtype=object)
V = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(U.shape):
    offsets[ind] = rng.random((ind[0]+1, 2)) * 6
    U[ind] = rng.random(ind[0]+1) * 2
    V[ind] = rng.random(ind[0]+1) * 2

m = hs.plot.markers.Arrows(
    offsets,
    U,
    V,
)

s.plot()
s.add_marker(m)
```



.



sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 0.820 seconds)

22.1.8 Circle Markers

Create a signal

```
import hyperspy.api as hs
import numpy as np

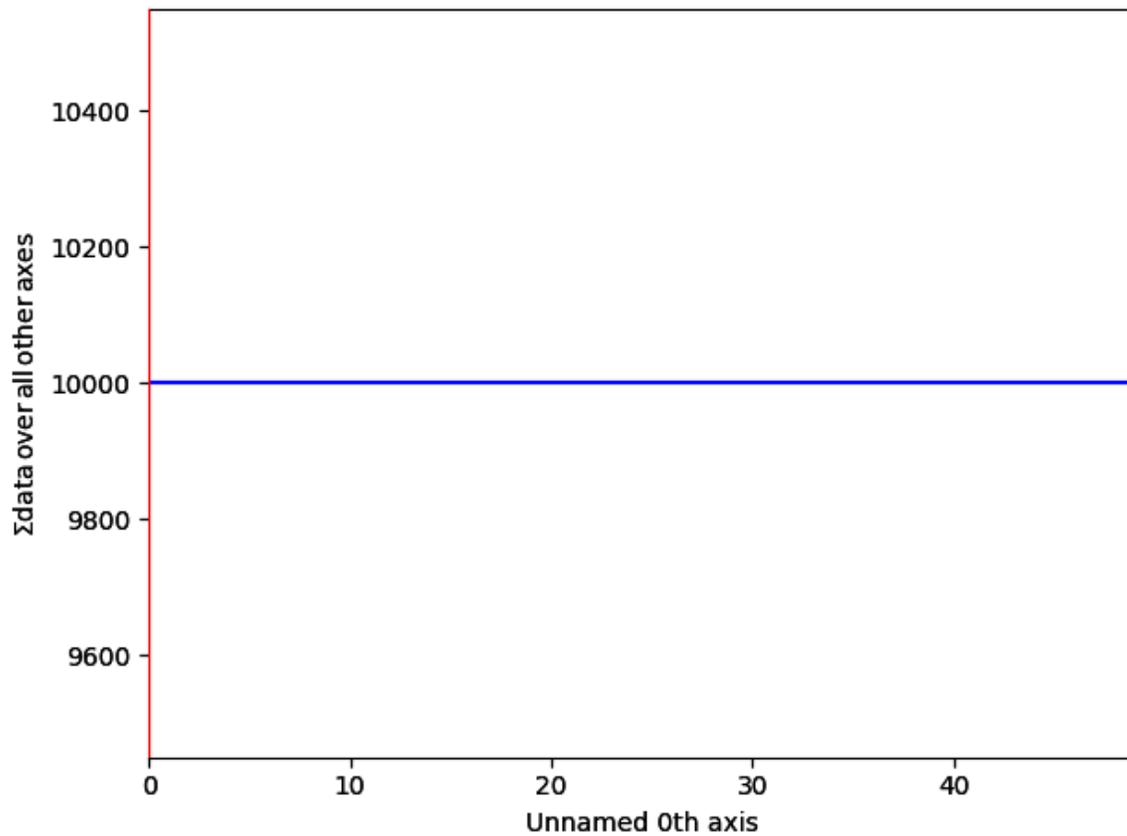
# Create a Signal2D with 1 navigation dimension
rng = np.random.default_rng(0)
data = np.ones((50, 100, 100))
s = hs.signals.Signal2D(data)
```

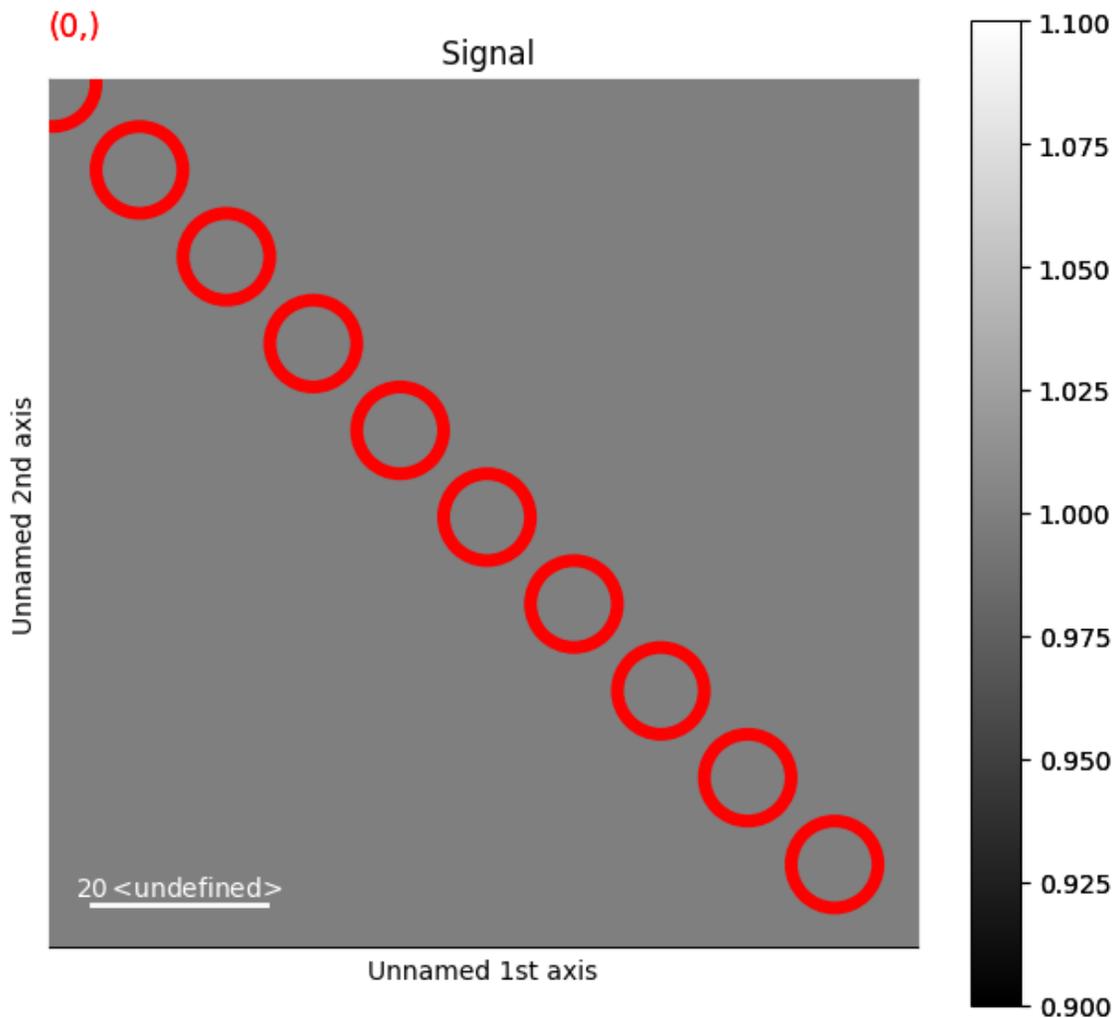
This first example shows how to draw static circles

```
# Define the position of the circles (start at (0, 0) and increment by 10)
offsets = np.array([np.arange(0, 100, 10)]*2).T

m = hs.plot.markers.Circles(
    sizes=10,
    offsets=offsets,
    edgecolor='r',
    linewidth=5,
)

s.plot()
s.add_marker(m)
```





Dynamic Circle Markers

This second example shows how to draw dynamic circles whose position and radius change depending on the navigation position

```
s2 = hs.signals.Signal2D(data)

offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
sizes = np.empty(s.axes_manager.navigation_shape, dtype=object)

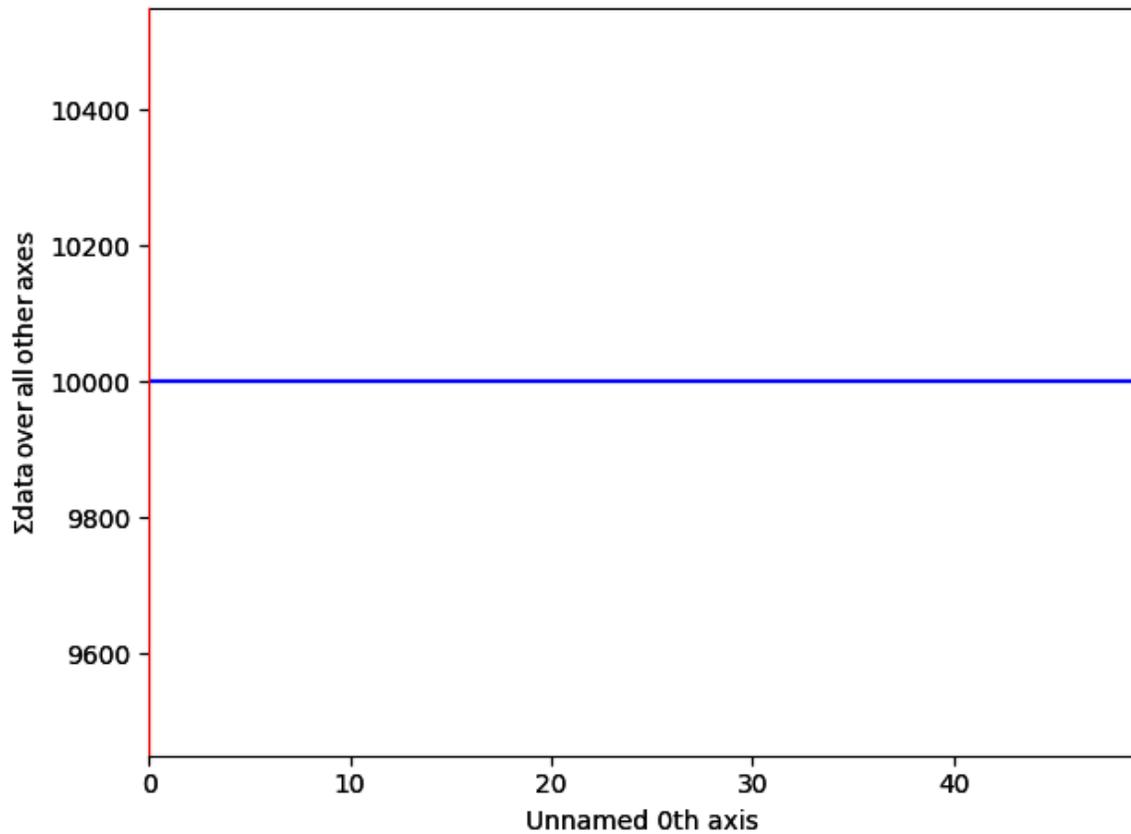
for ind in np.ndindex(offsets.shape):
    offsets[ind] = rng.random((5, 2)) * 100
    sizes[ind] = rng.random((5, )) * 10

m = hs.plot.markers.Circles(
    sizes=sizes,
    offsets=offsets,
    edgecolor='r',
```

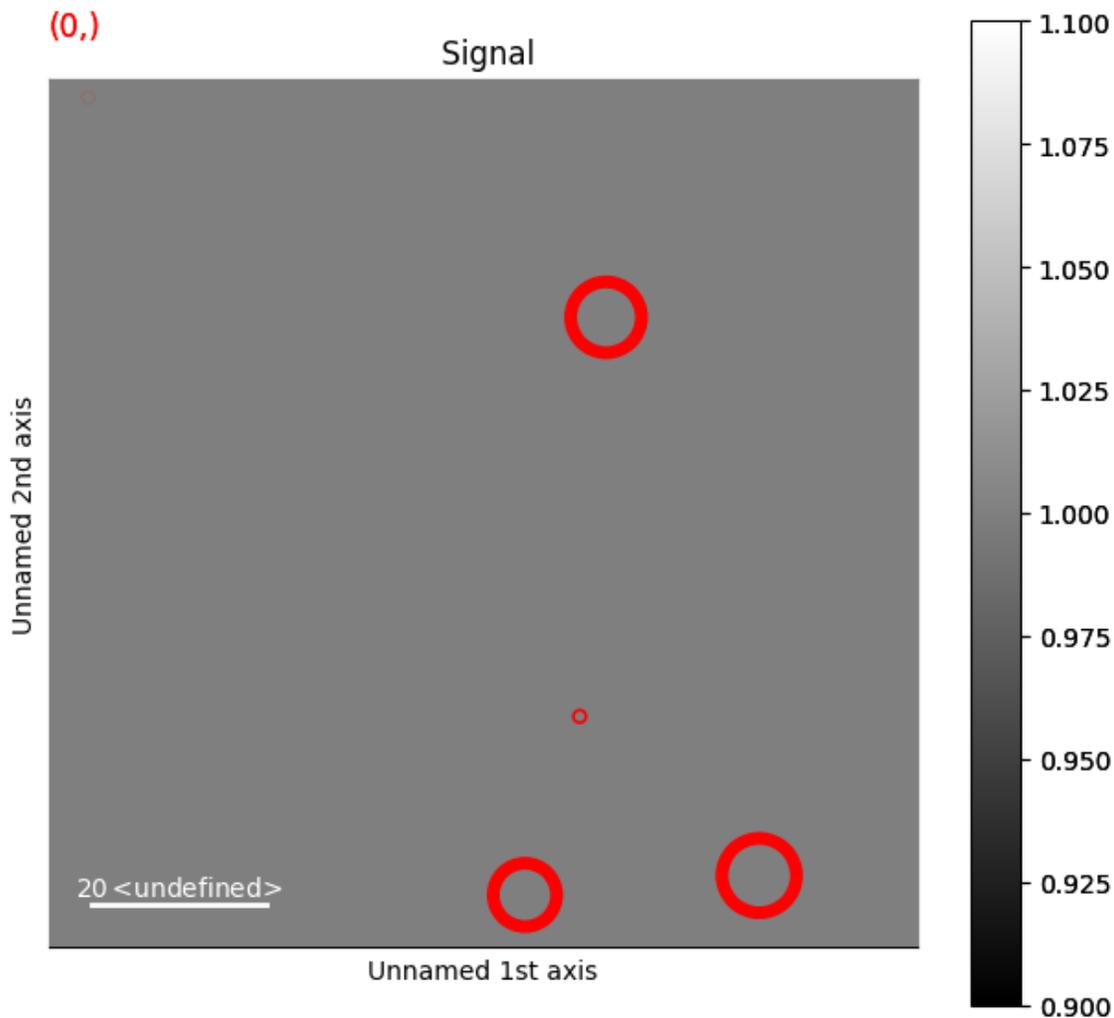
(continues on next page)

(continued from previous page)

```
linewidth=5,  
)  
s2.plot()  
s2.add_marker(m)
```



•



sphinx_gallery_thumbnail_number = 4

Total running time of the script: (0 minutes 1.291 seconds)

22.1.9 Filled Circle Markers

Create a signal

```
import hyperspy.api as hs
import matplotlib as mpl
import numpy as np

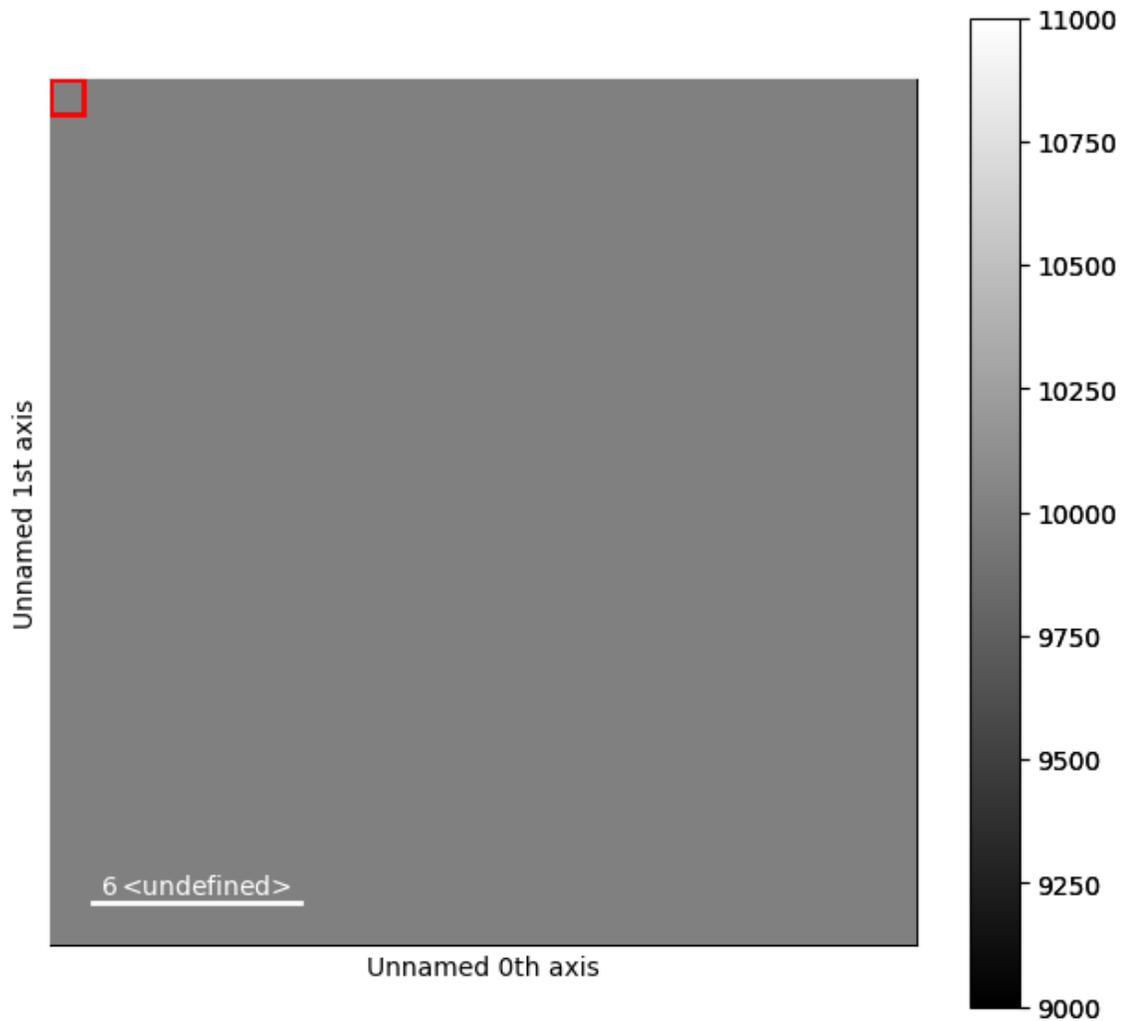
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

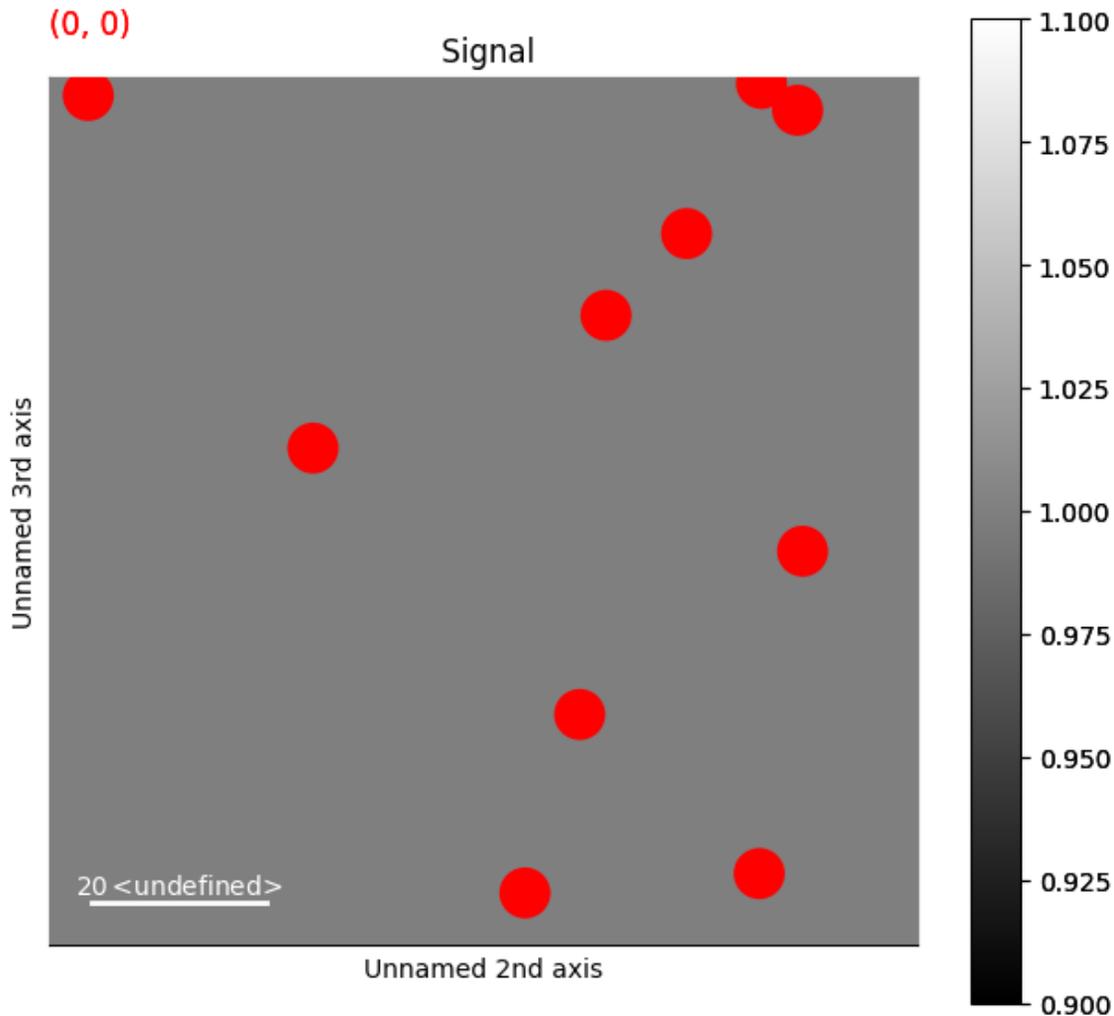
This first example shows how to draw static filled circles

```
# Define the position of the circles
offsets = rng.random((10, 2)) * 100

m = hs.plot.markers.Points(
    sizes=20,
    offsets=offsets,
    facecolors="red",
)

s.plot()
s.add_marker(m)
```





Dynamic Filled Circle Markers

This second example shows how to draw dynamic filled circles, whose size, color and position change depending on the navigation position

```
s2 = hs.signals.Signal2D(data)

offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
sizes = np.empty(s.axes_manager.navigation_shape, dtype=object)
colors = list(mpl.colors.TABLEAU_COLORS.values())[:10]

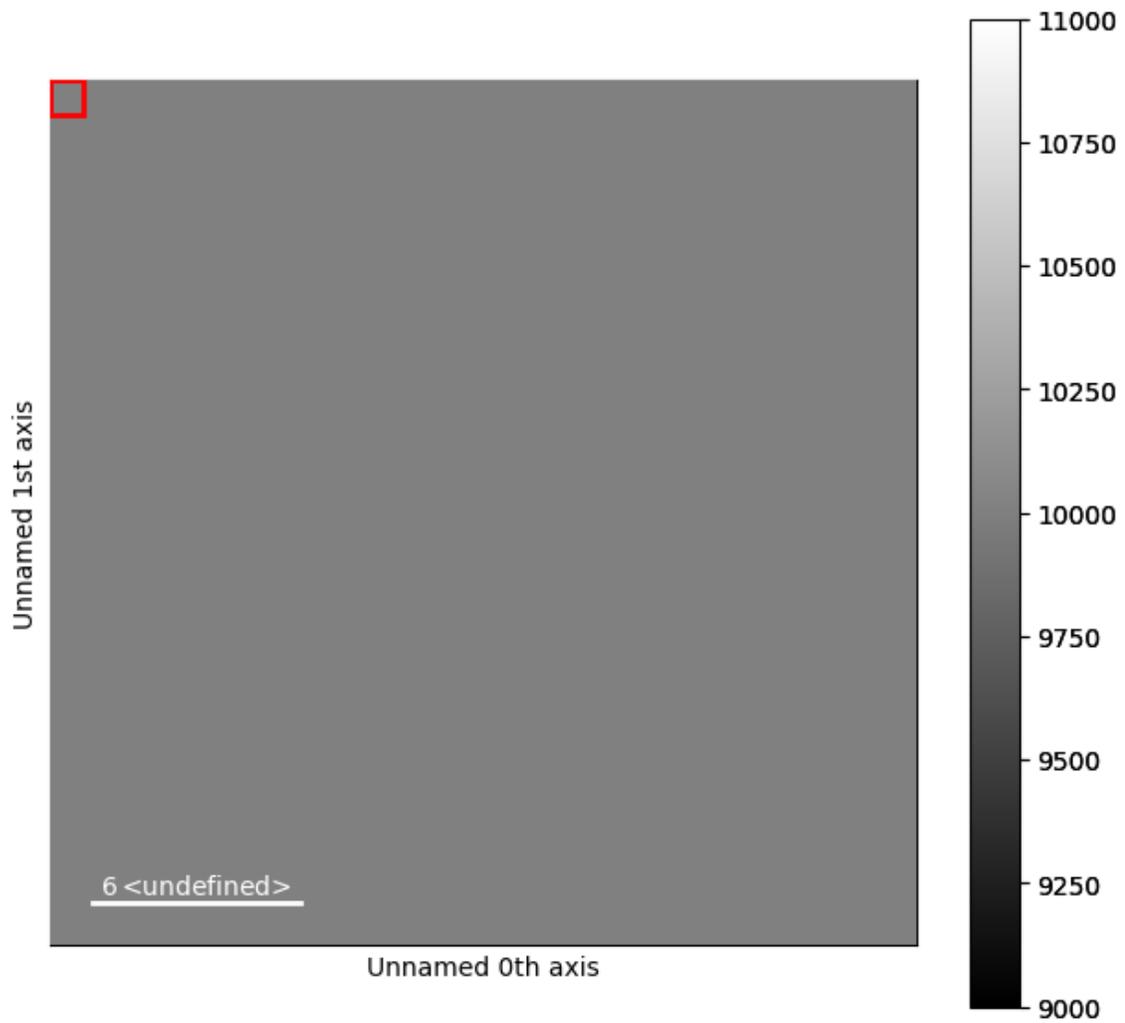
for ind in np.ndindex(offsets.shape):
    offsets[ind] = rng.random((10, 2)) * 100
    sizes[ind] = rng.random((10, )) * 50

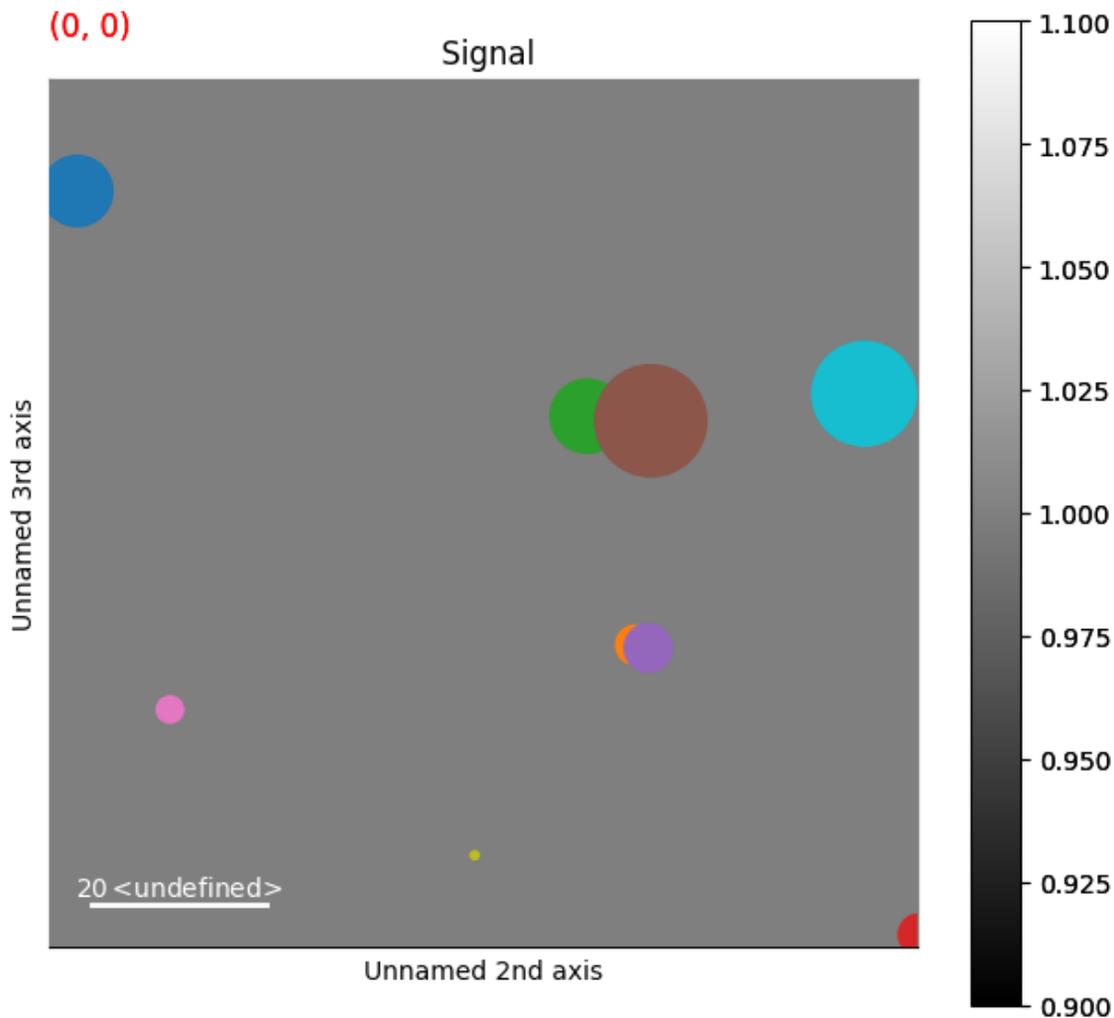
m = hs.plot.markers.Points(
    sizes=sizes,
    offsets=offsets,
```

(continues on next page)

(continued from previous page)

```
facecolors=colors,  
)  
s2.plot()  
s2.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.552 seconds)

22.1.10 Rotation of markers

This example shows how markers are rotated.

Create a signal

```
import hyperspy.api as hs
import numpy as np

# Create a Signal2D
data = np.ones([100, 100])
s = hs.signals.Signal2D(data)

num = 2
angle = 25
```

(continues on next page)

```
color = ["tab:orange", "tab:blue"]
```

Create the markers, the first and second elements are at 0 and 20 degrees

```
# Define the position of the markers
offsets = np.array([20*np.ones(num)]*2).T
angles = np.arange(0, angle*num, angle)

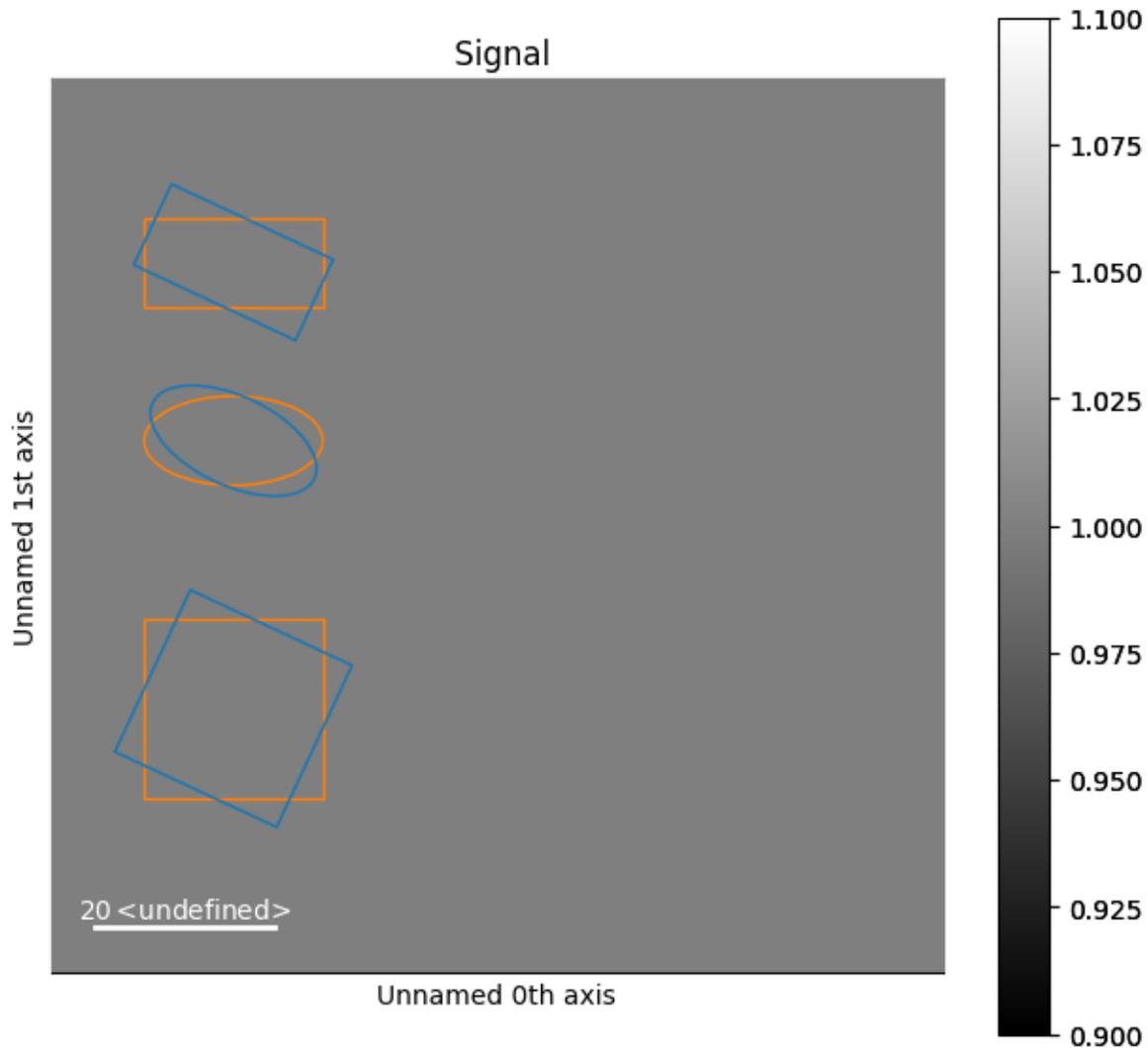
m1 = hs.plot.markers.Rectangles(
    offsets=offsets,
    widths=np.ones(num)*20,
    heights=np.ones(num)*10,
    angles=angles,
    facecolor='none',
    edgecolor=color,
)

m2 = hs.plot.markers.Ellipses(
    offsets=offsets + np.array([0, 20]),
    widths=np.ones(num)*20,
    heights=np.ones(num)*10,
    angles=angles,
    facecolor='none',
    edgecolor=color,
)

m3 = hs.plot.markers.Squares(
    offsets=offsets + np.array([0, 50]),
    widths=np.ones(num)*20,
    angles=angles,
    facecolor='none',
    edgecolor=color,
)
```

Plot the signals and add all the markers

```
s.plot()
s.add_marker([m1, m2, m3])
```



sphinx_gallery_thumbnail_number = 1

Total running time of the script: (0 minutes 0.347 seconds)

22.1.11 Add/Remove items from existing Markers

This example shows how to add or remove marker from an existing collection. This is done by setting the parameters (offsets, sizes, etc.) of the collection.

Create a signal

```
import hyperspy.api as hs
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.arange(15*100*100).reshape((15, 100, 100))
s = hs.signals.Signal2D(data)
```

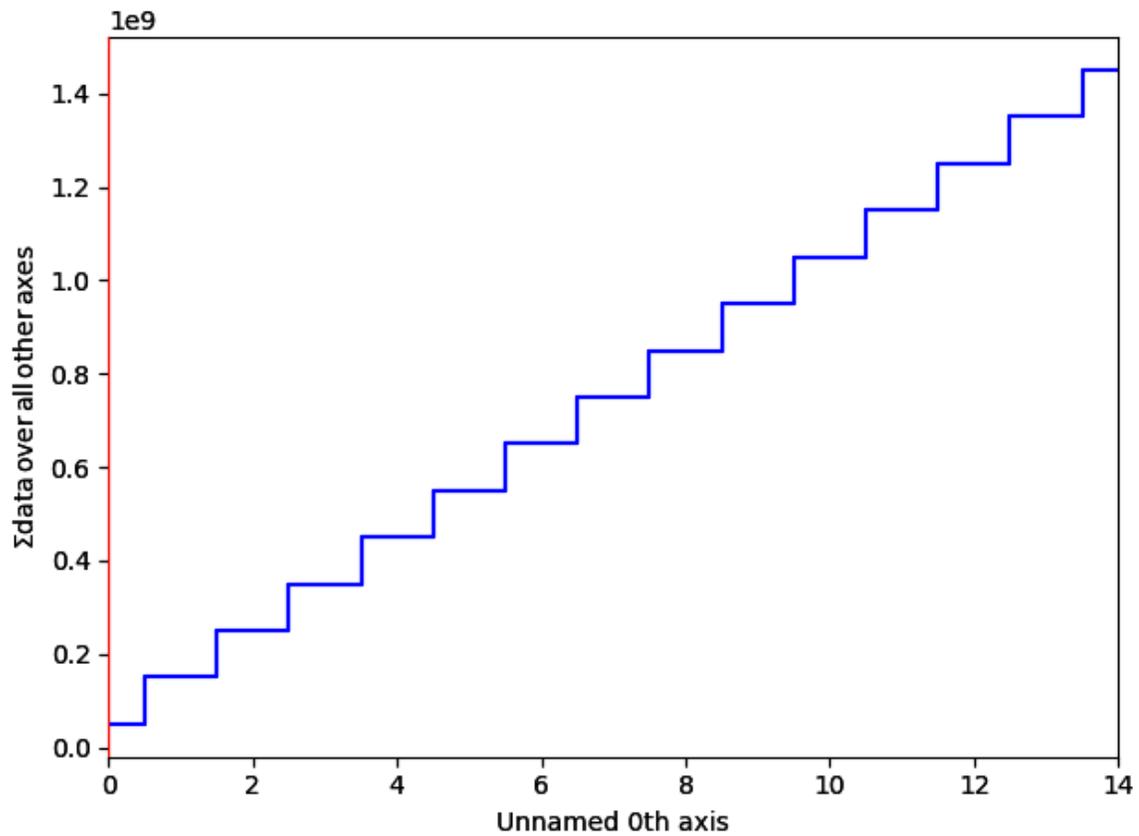
Create text marker

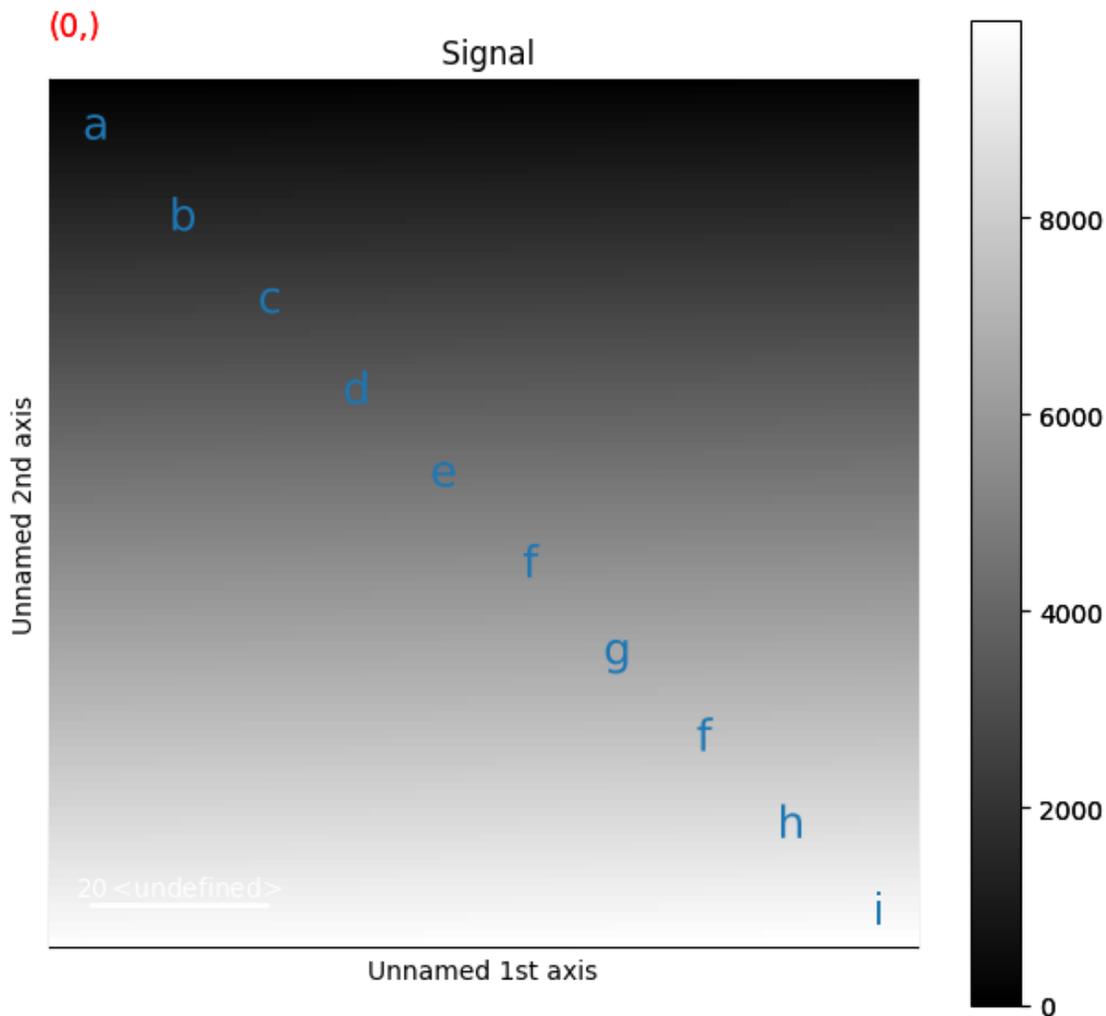
```
# Define the position of the texts
offsets = np.stack([np.arange(0, 100, 10)]*2).T + np.array([5,]*2)
texts = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'f', 'h', 'i'])

m = hs.plot.markers.Texts(
    offsets=offsets,
    texts=texts,
    sizes=3,
)

print(f'Number of markers is {len(m)}.')

s.plot()
s.add_marker(m)
```





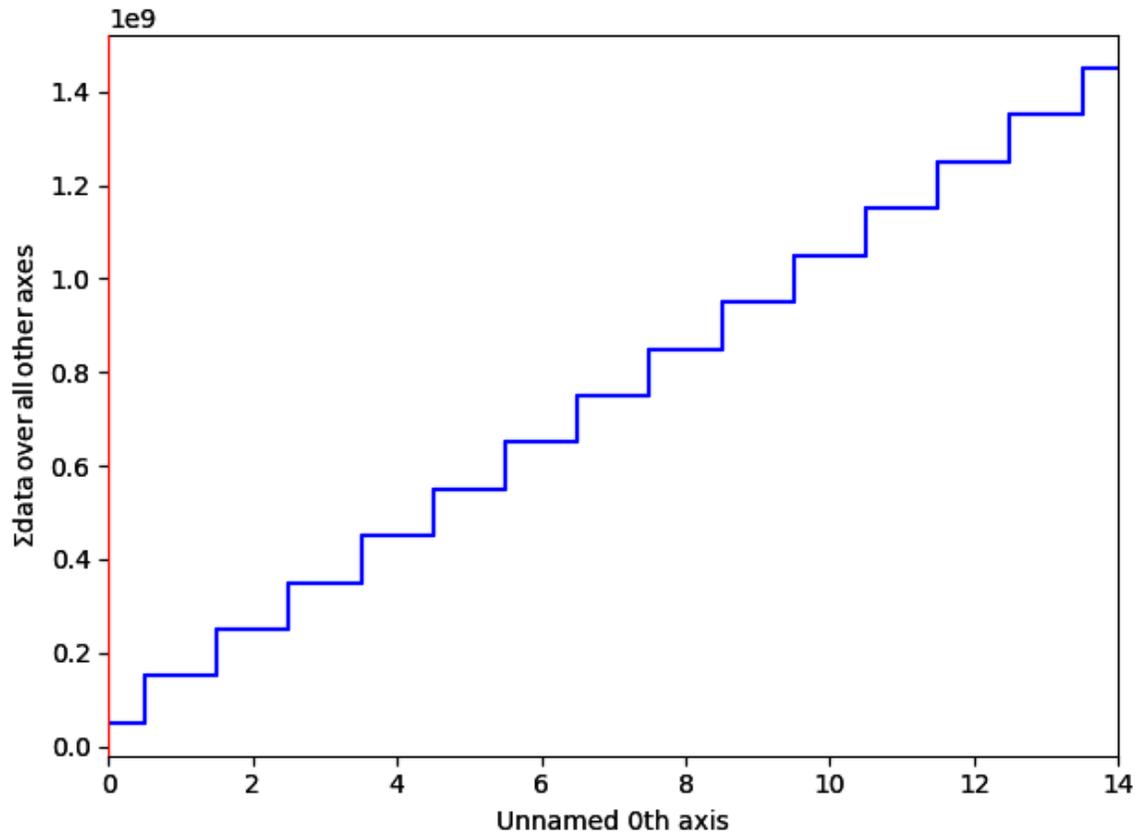
Number of markers is 10.

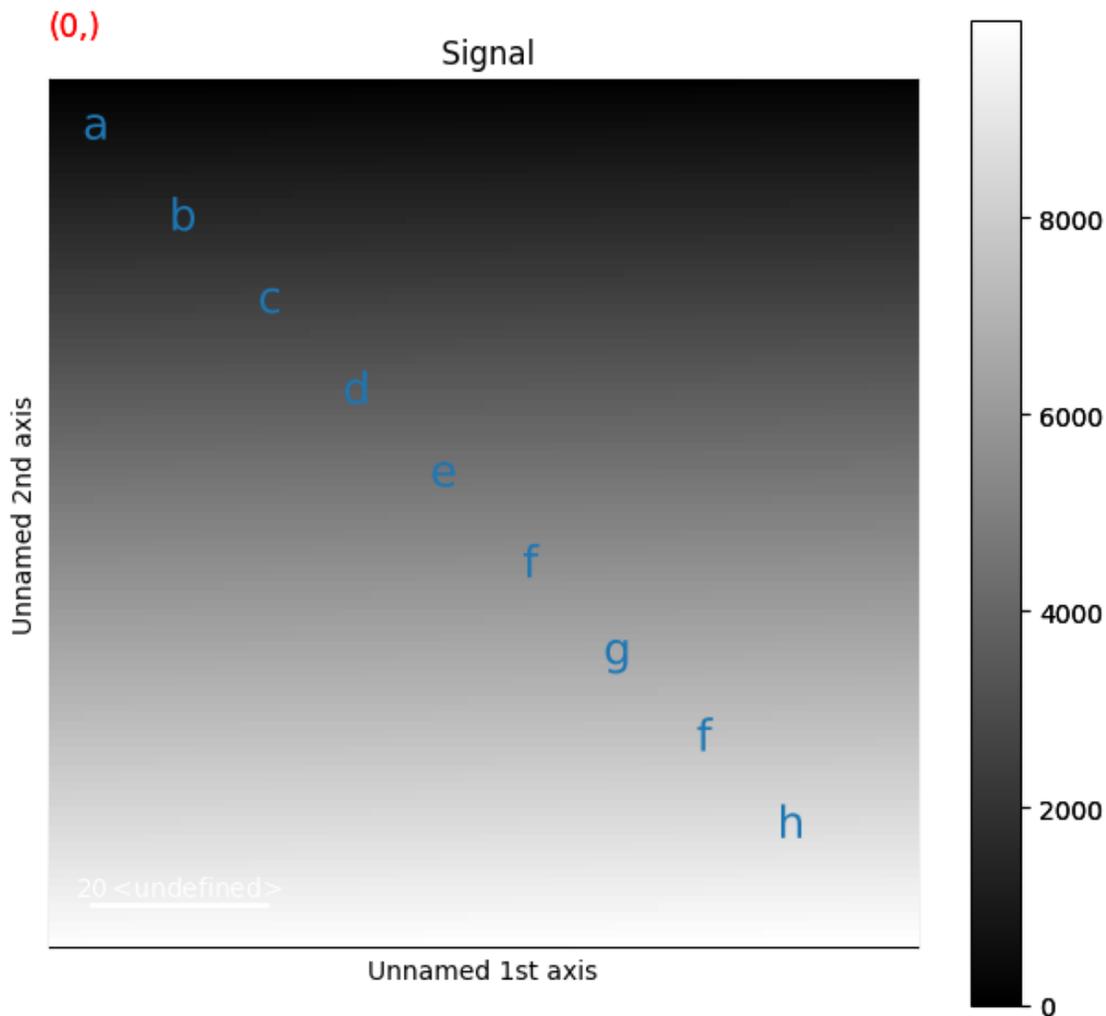
Remove the last text of the collection

```
# Set new texts and offsets parameters with one less item
m.remove_items(indices=-1)

print(f'Number of markers is {len(m)} after removing one marker.')

s.plot()
s.add_marker(m)
```



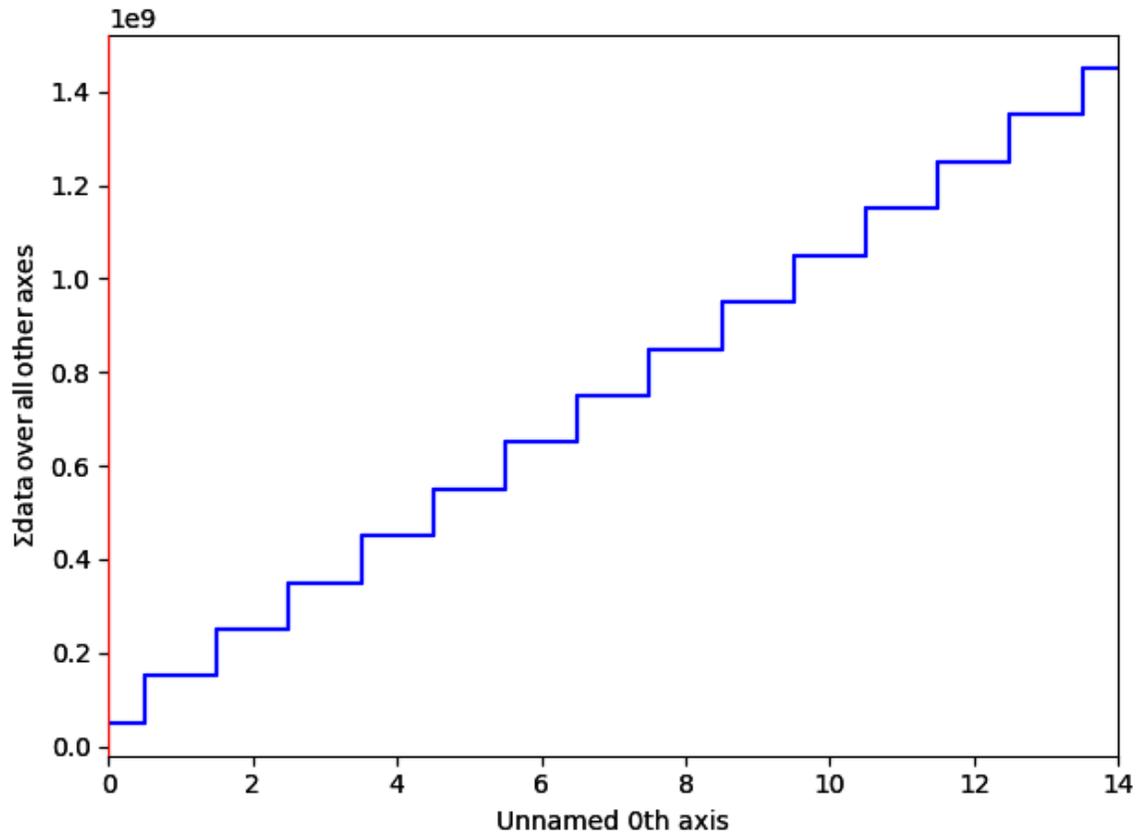


Number of markers is 9 after removing one marker.

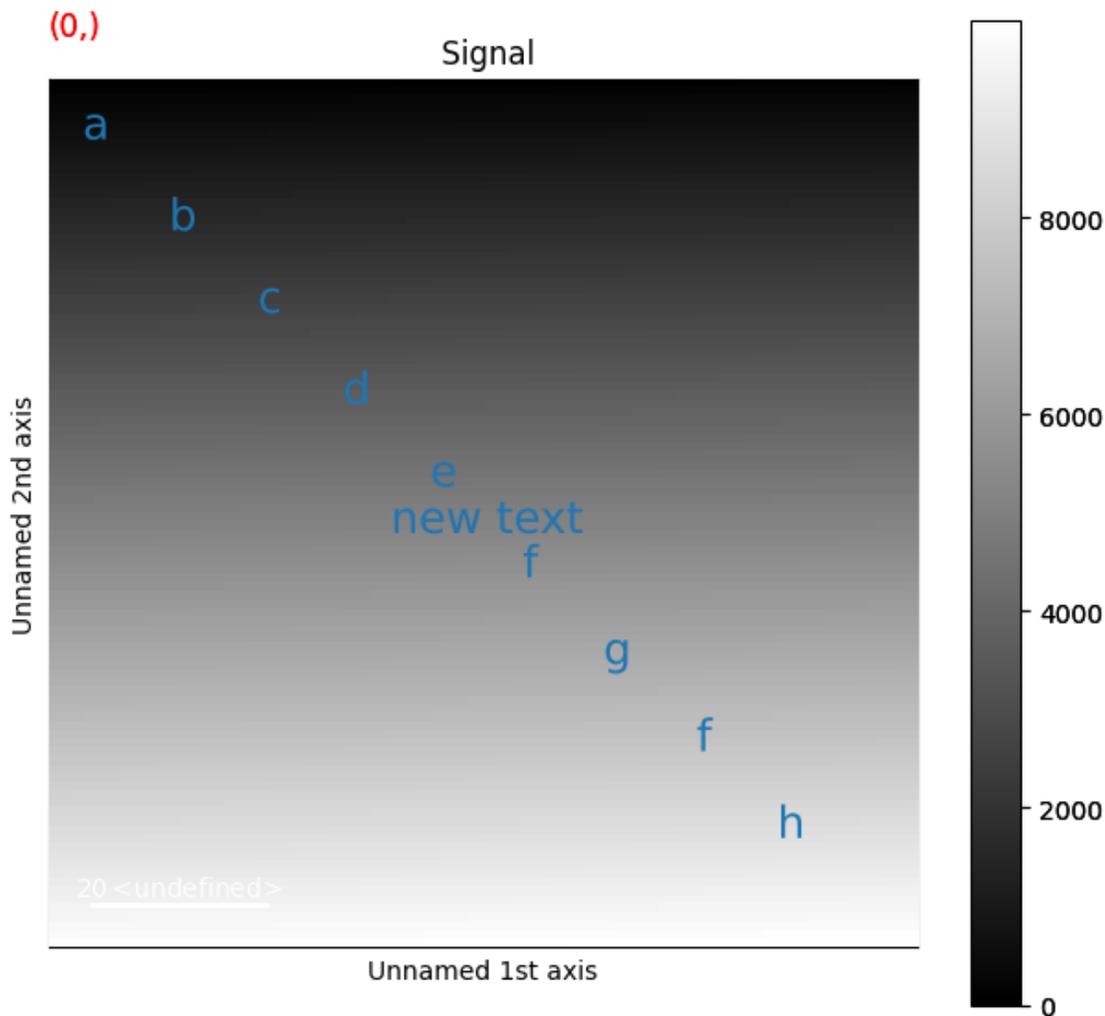
Add another text of the collection

```
# Define the position in the middle of the axes
m.add_items(offsets=np.array([[50, 50]]), texts=np.array(["new text"]))
print(f'Number of markers is {len(m)} after adding the text {texts[-1]}')

s.plot()
s.add_marker(m)
```



.



Number of markers is 10 after adding the text i.

sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.983 seconds)

22.1.12 Polygon Markers

Create a signal

```
import hyperspy.api as hs
import matplotlib.pyplot as plt
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

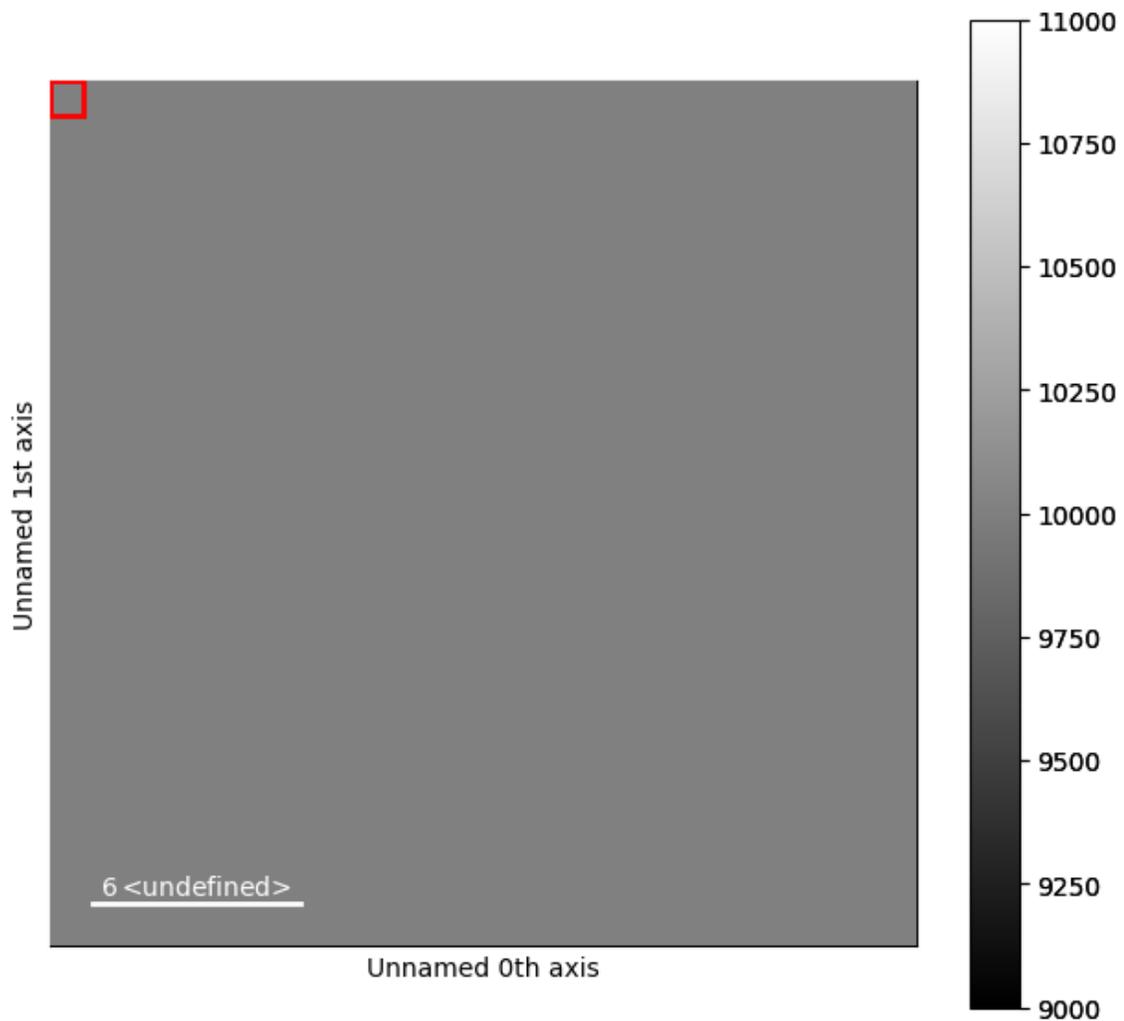
This first example shows how to draw static polygon markers using the matplotlib PolygonCollection

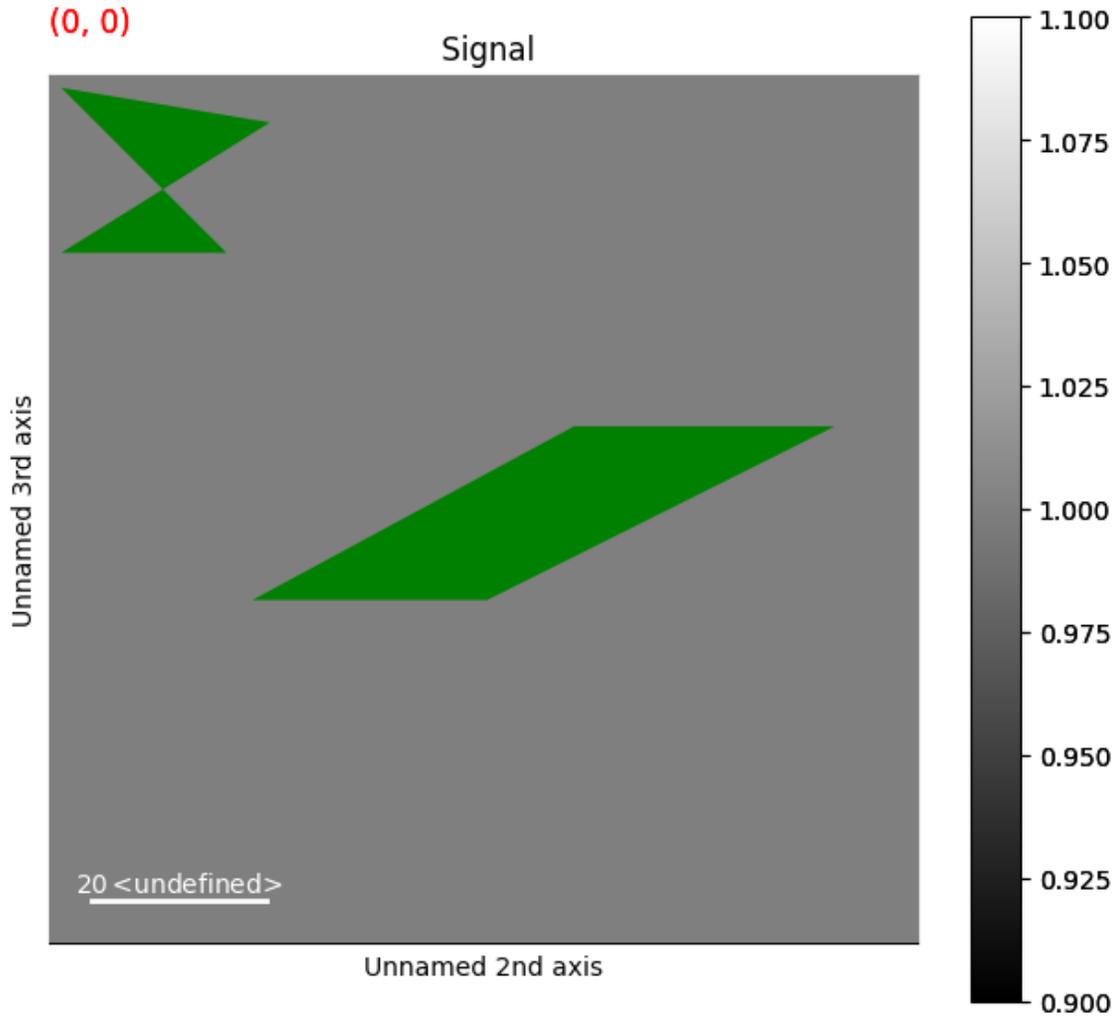
```
# Define the vertexes of the polygons
# poylgon1: [[x0, y0], [x1, y1], [x2, y2], [x3, x3]]
# poylgon2: [[x0, y0], [x1, y1], [x2, y2]]
# ...
poylgon1 = [[1, 1], [20, 20], [1, 20], [25, 5]]
poylgon2 = [[50, 60], [90, 40], [60, 40], [23, 60]]

verts = [poylgon1, poylgon2]

m = hs.plot.markers.Polygons(
    verts=verts,
    linewidth=3,
    facecolors=('g',),
)

s.plot()
s.add_marker(m)
```





Dynamic Polygon Markers

This example shows how to draw dynamic polygon markers, whose position depends on the navigation coordinates

```
verts = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(verts.shape):
    verts[ind] = rng.random((10, 4, 2)) * 100

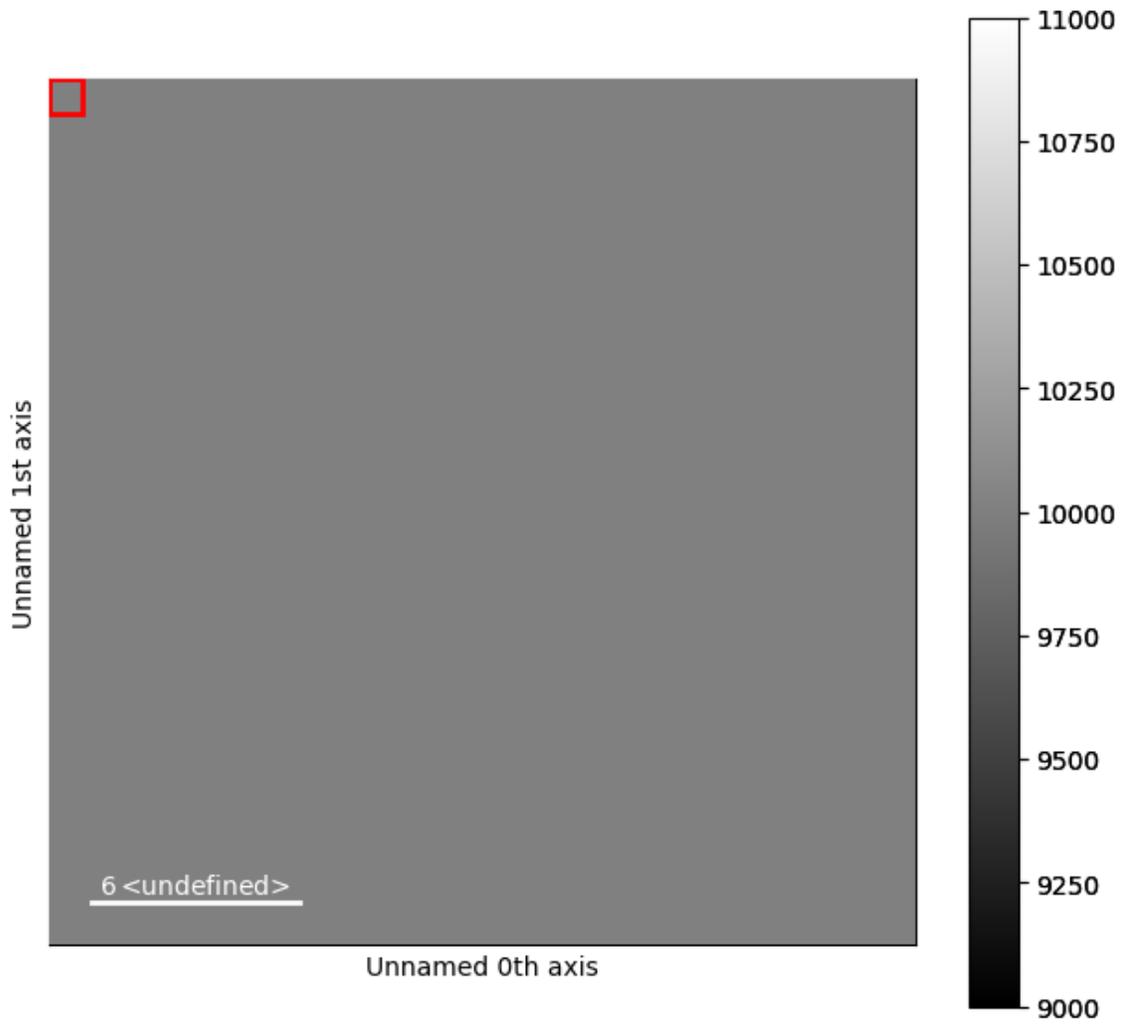
# Get list of colors
colors = list(plt.rcParams['axes.prop_cycle'].by_key()['color'])

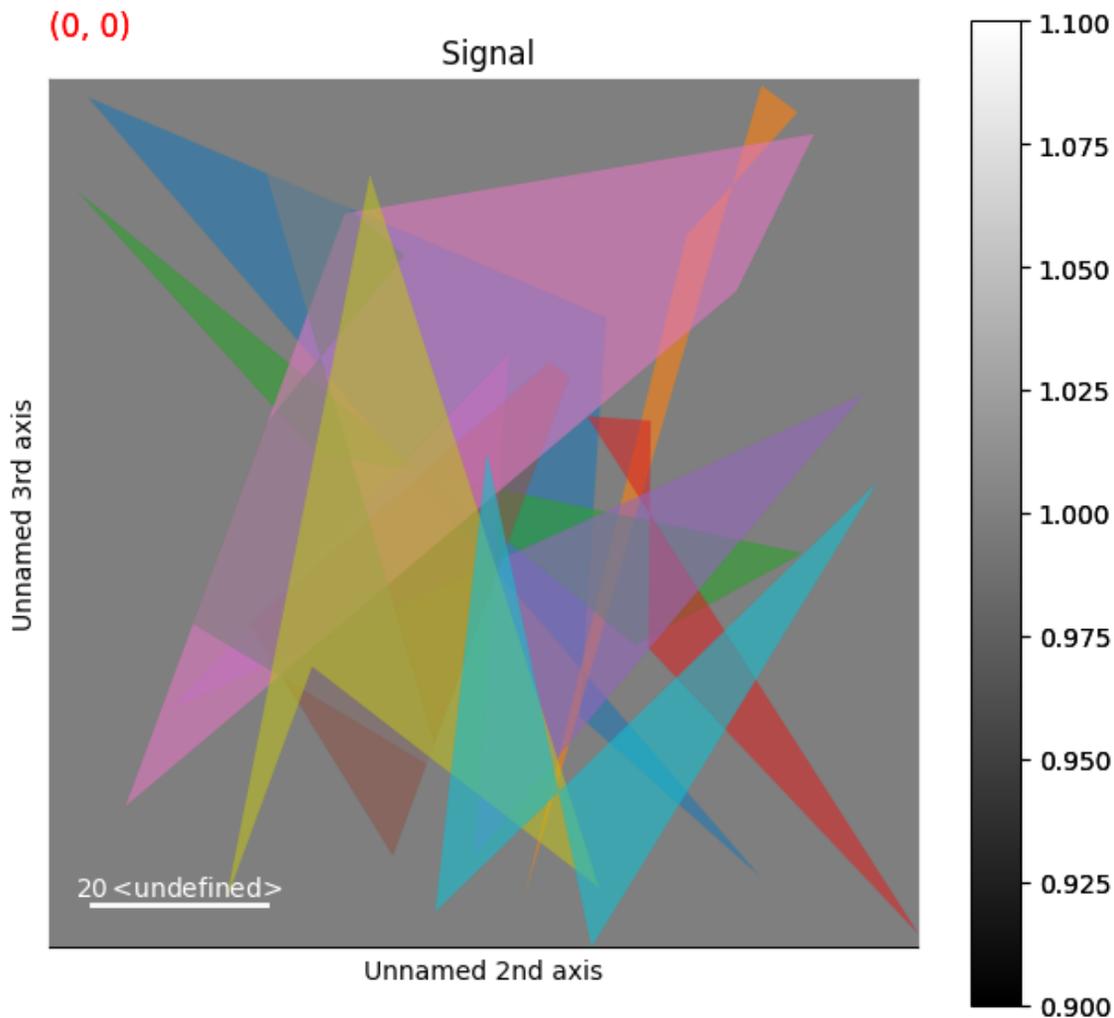
m = hs.plot.markers.Polygons(
    verts=verts,
    facecolors=colors,
    linewidth=3,
    alpha=0.6
)
s.plot()
```

(continues on next page)

(continued from previous page)

```
s.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.571 seconds)

22.1.13 Ellipse markers

Create a signal

```
import hyperspy.api as hs
import numpy as np

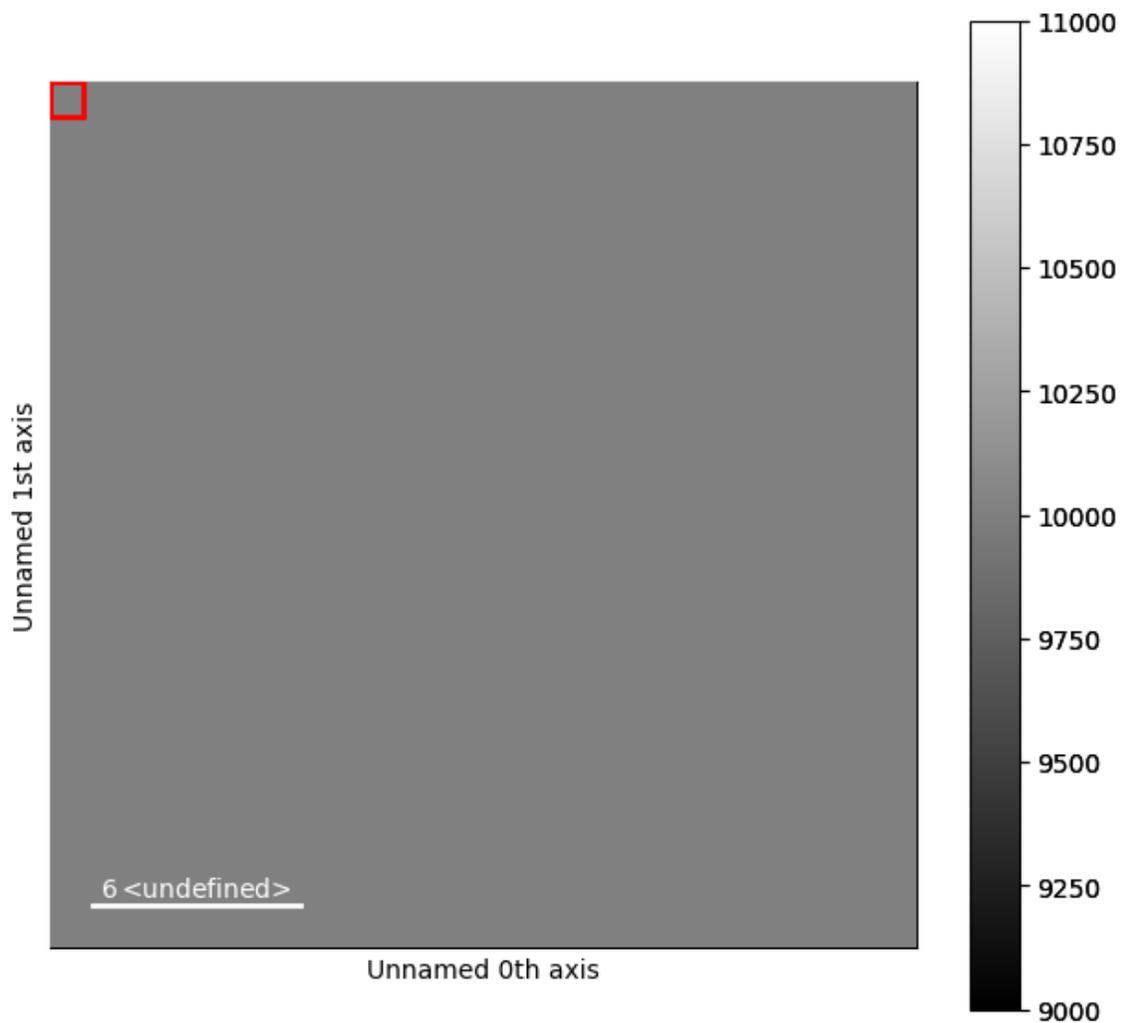
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

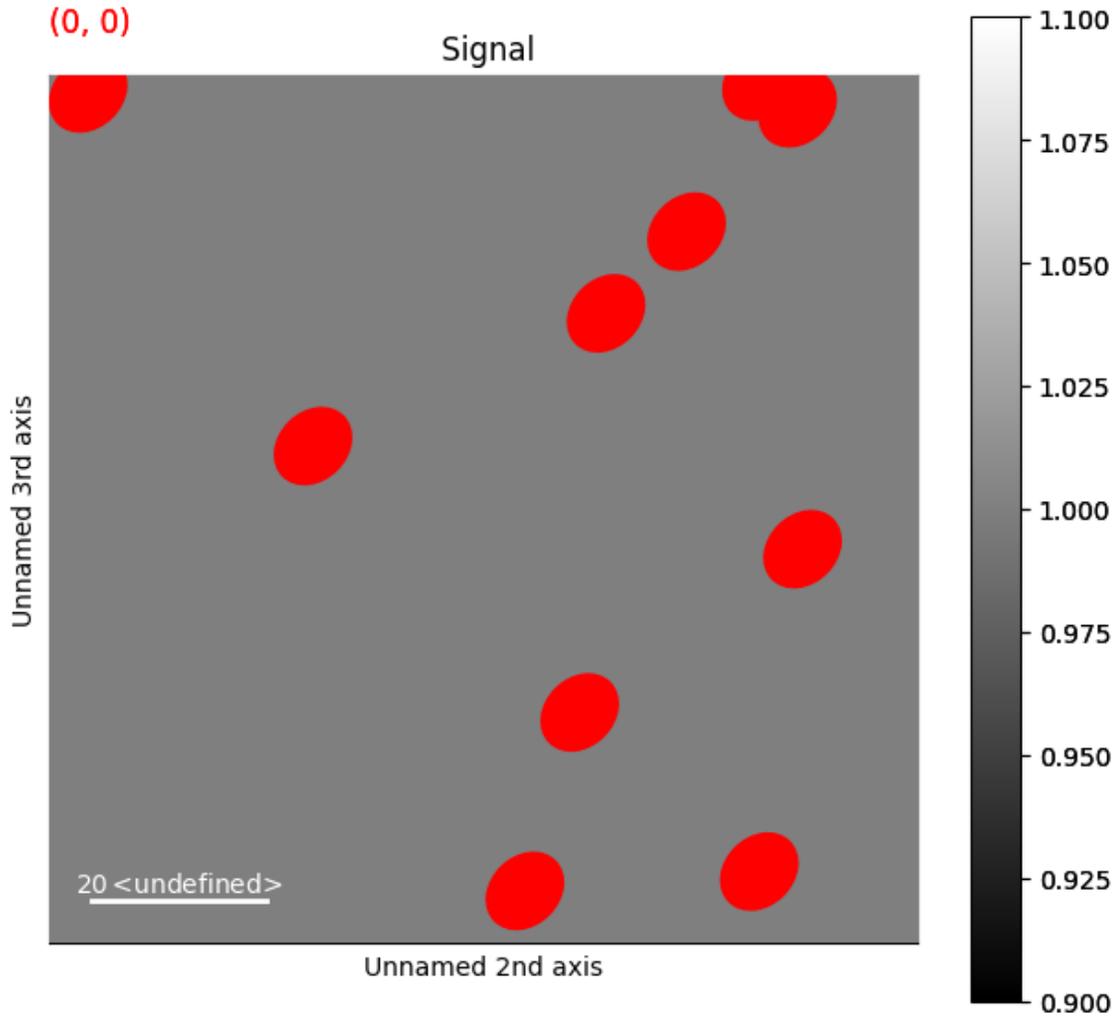
This first example shows how to draw static ellipses

```
# Define the position of the ellipses
offsets = rng.random((10, 2)) * 100

m = hs.plot.markers.Ellipses(
    widths=(8,),
    heights=(10,),
    angles=(45,),
    offsets=offsets,
    facecolor="red",
)

s.plot()
s.add_marker(m)
```





Dynamic Ellipse Markers

This first example shows how to draw dynamic ellipses, whose position, widths heights and angles depends on the navigation coordinates

```
s2 = hs.signals.Signal2D(data)

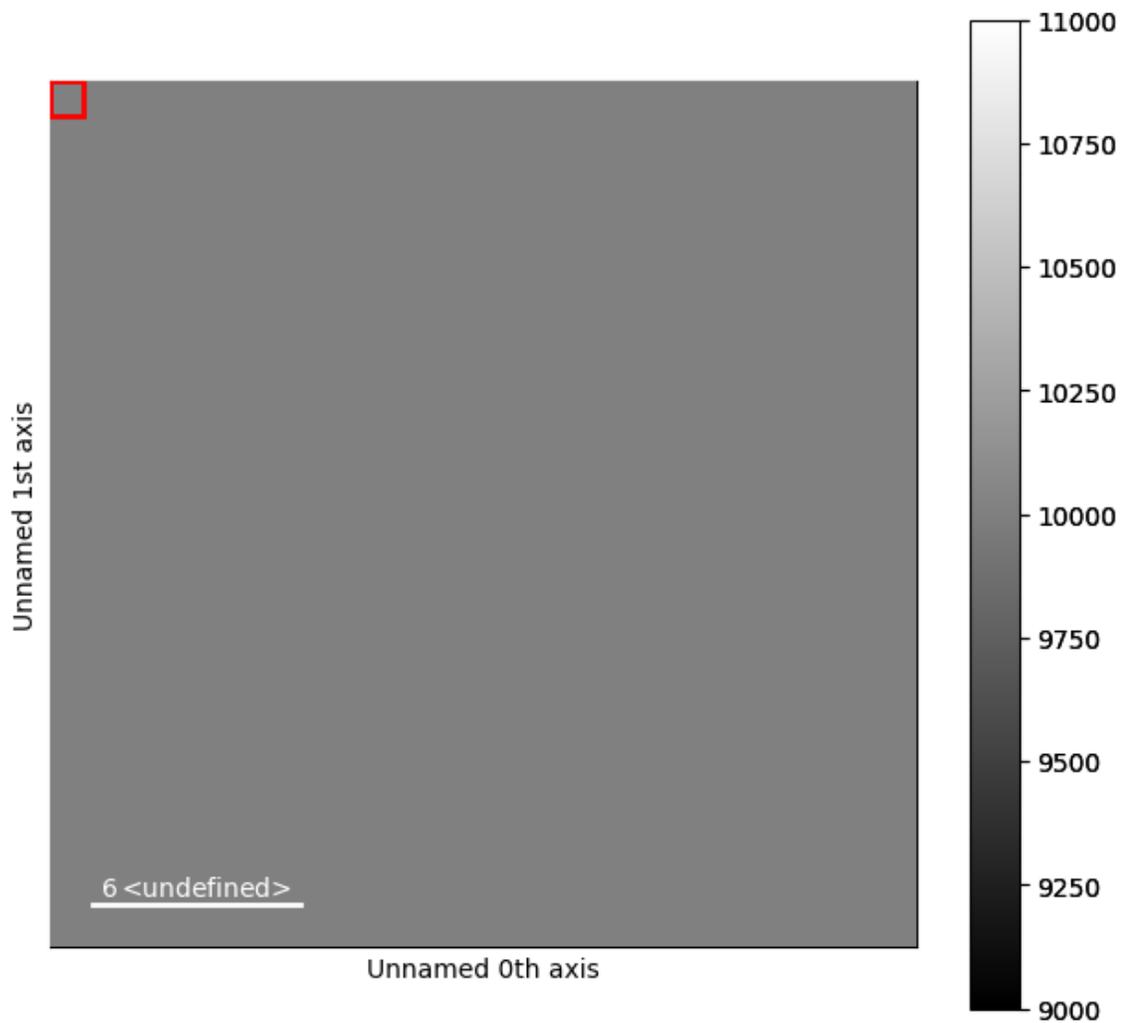
widths = np.empty(s.axes_manager.navigation_shape, dtype=object)
heights = np.empty(s.axes_manager.navigation_shape, dtype=object)
angles = np.empty(s.axes_manager.navigation_shape, dtype=object)
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)

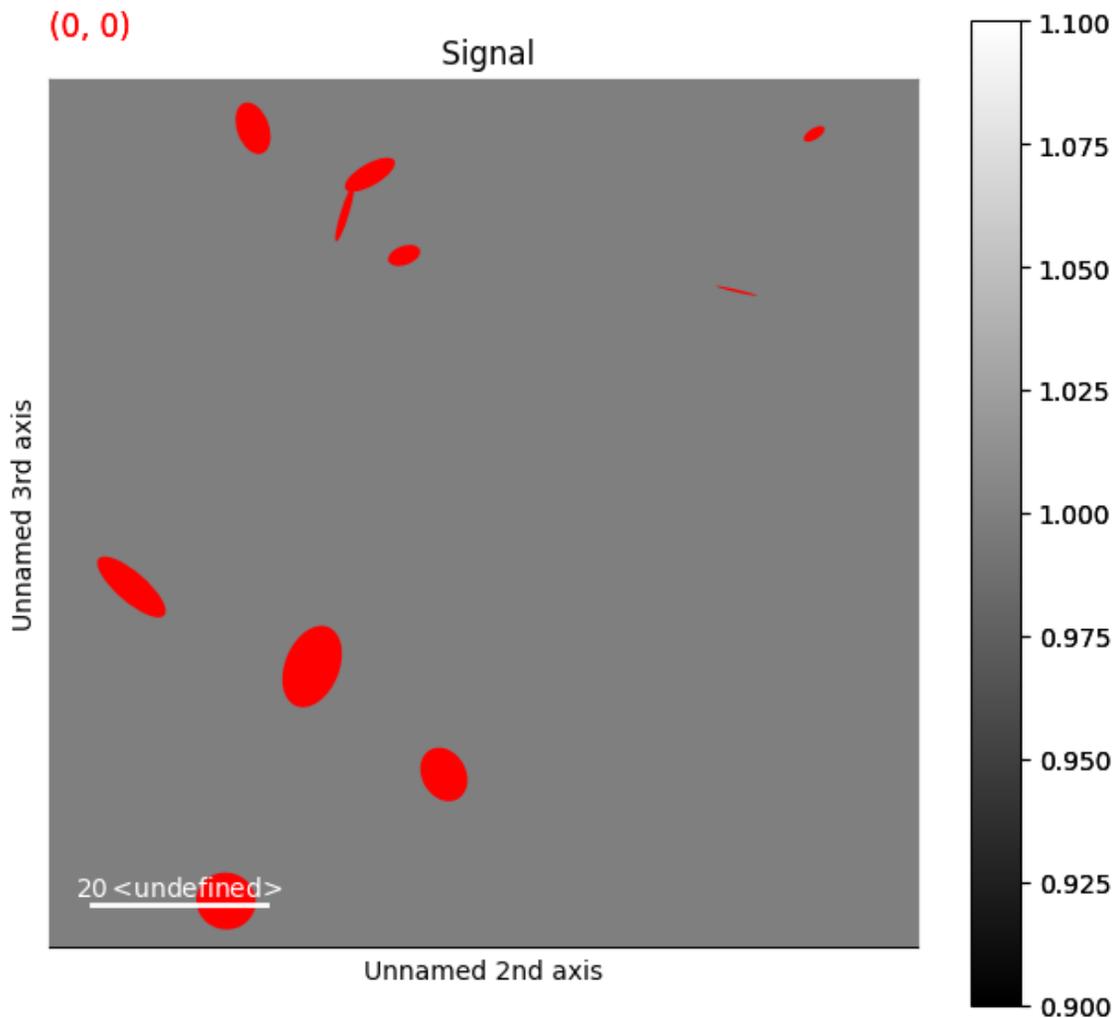
for index in np.ndindex(offsets.shape):
    widths[index] = rng.random((10, )) * 10
    heights[index] = rng.random((10, )) * 7
    angles[index] = rng.random((10, )) * 180
    offsets[index] = rng.random((10, 2)) * 100
```

(continues on next page)

(continued from previous page)

```
m = hs.plot.markers.Ellipses(  
    widths=widths,  
    heights=heights,  
    angles=angles,  
    offsets=offsets,  
    facecolor="red",  
    )  
  
s2.plot()  
s2.add_marker(m)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.563 seconds)

22.1.14 Square Markers

Create a signal

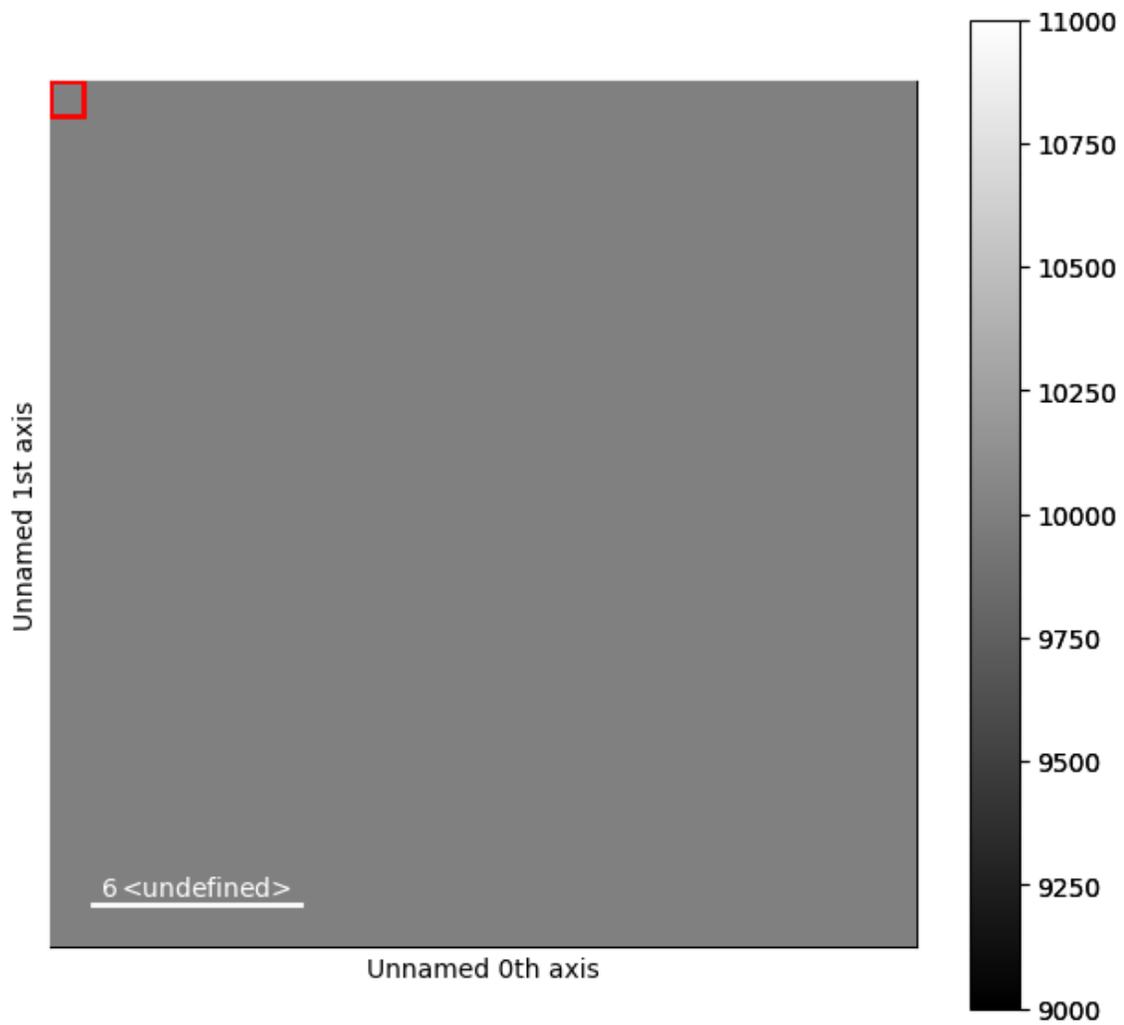
```
import hyperspy.api as hs
import numpy as np

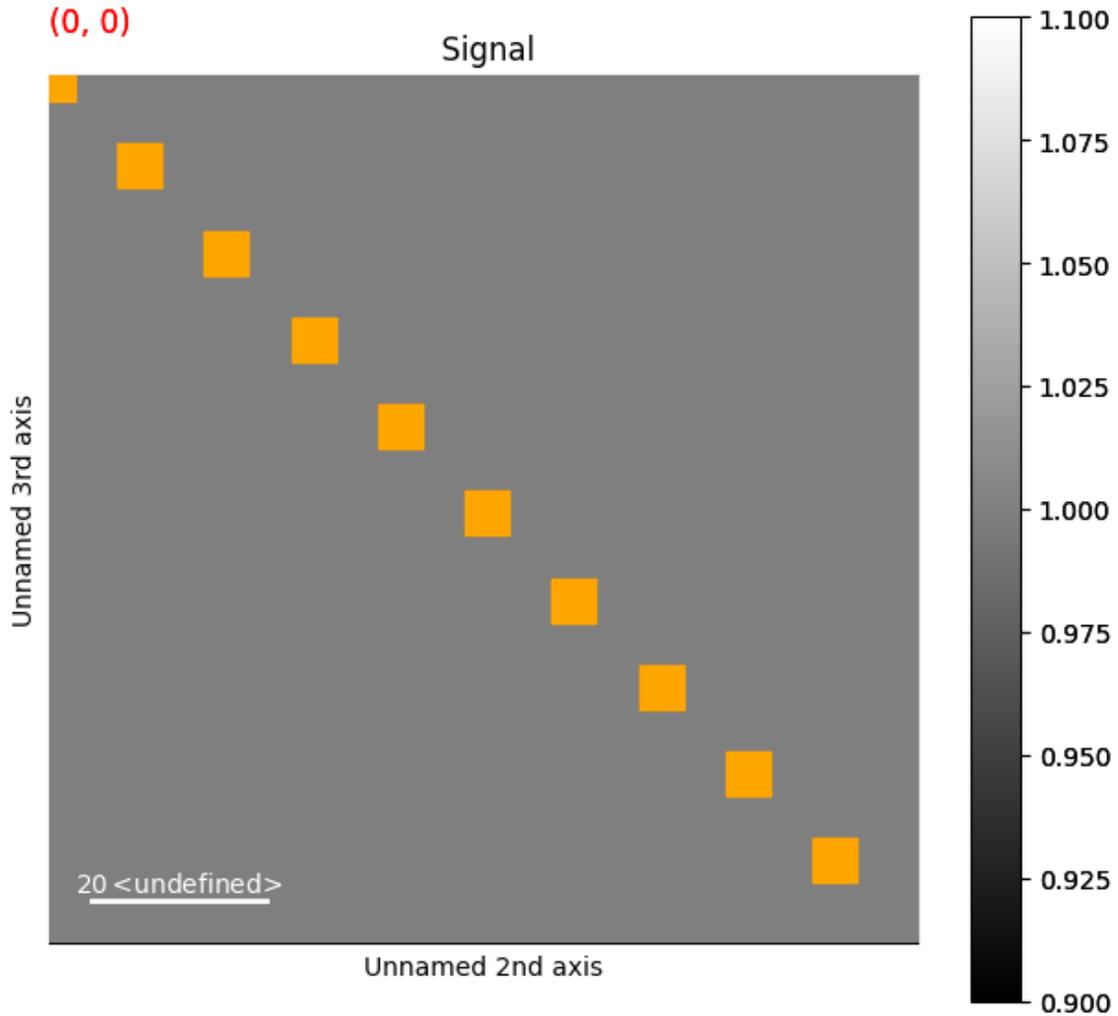
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

This first example shows how to draw static square markers

```
# Define the position of the squares (start at (0, 0) and increment by 10)
offsets = np.array([np.arange(0, 100, 10)]*2).T

m = hs.plot.markers.Squares(
    offsets=offsets,
    widths=(5,),
    angles=(0,),
    color="orange",
)
s.plot()
s.add_marker(m)
```





Dynamic Square Markers

This first example shows how to draw dynamic squares markers, whose position, widths and angles depends on the navigation coordinates

```
s2 = hs.signals.Signal2D(data)

widths = np.empty(s.axes_manager.navigation_shape, dtype=object)
heights = np.empty(s.axes_manager.navigation_shape, dtype=object)
angles = np.empty(s.axes_manager.navigation_shape, dtype=object)
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)

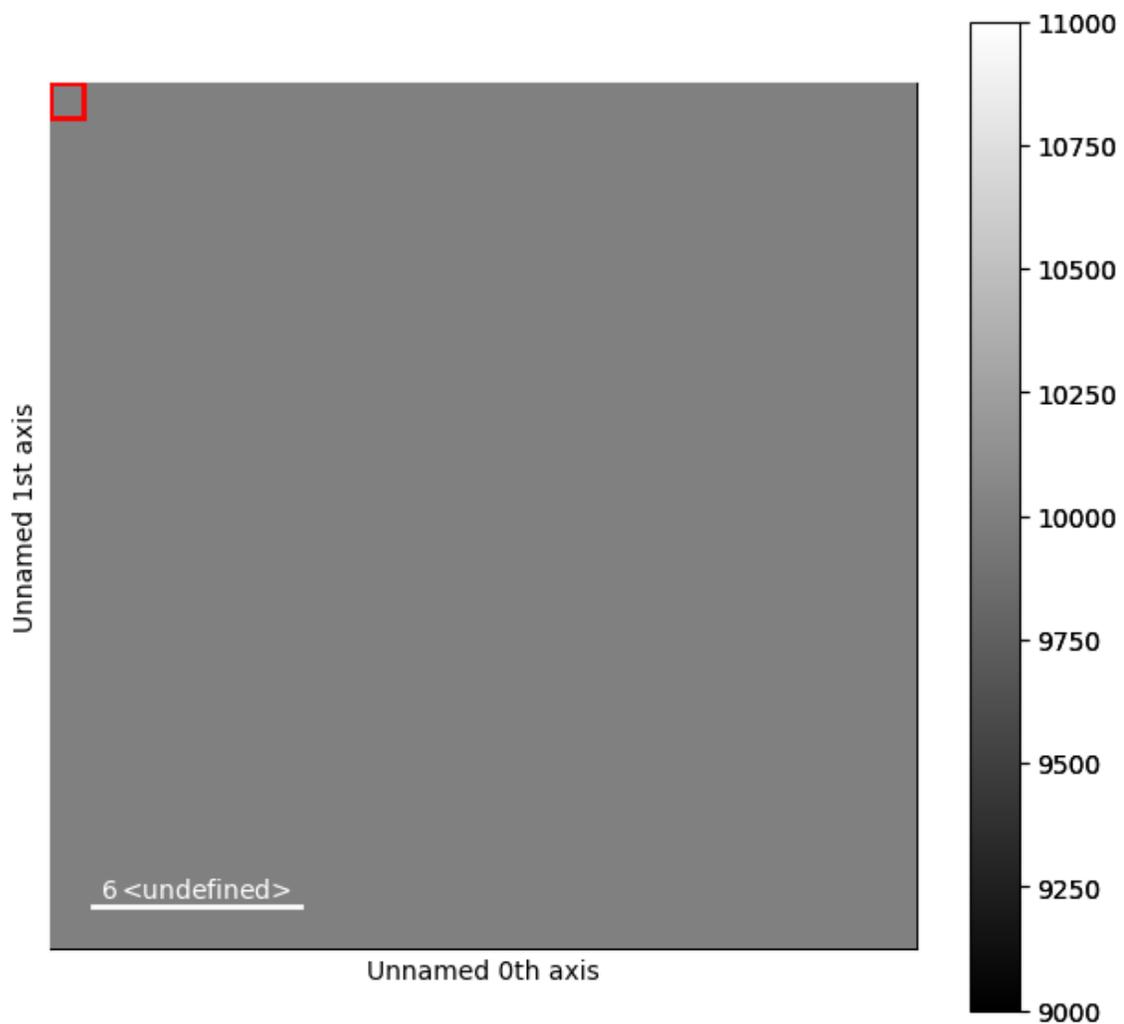
for index in np.ndindex(offsets.shape):
    widths[index] = rng.random((10, )) * 50
    heights[index] = rng.random((10, )) * 25
    angles[index] = rng.random((10, )) * 180
    offsets[index] = rng.random((10, 2)) * 100
```

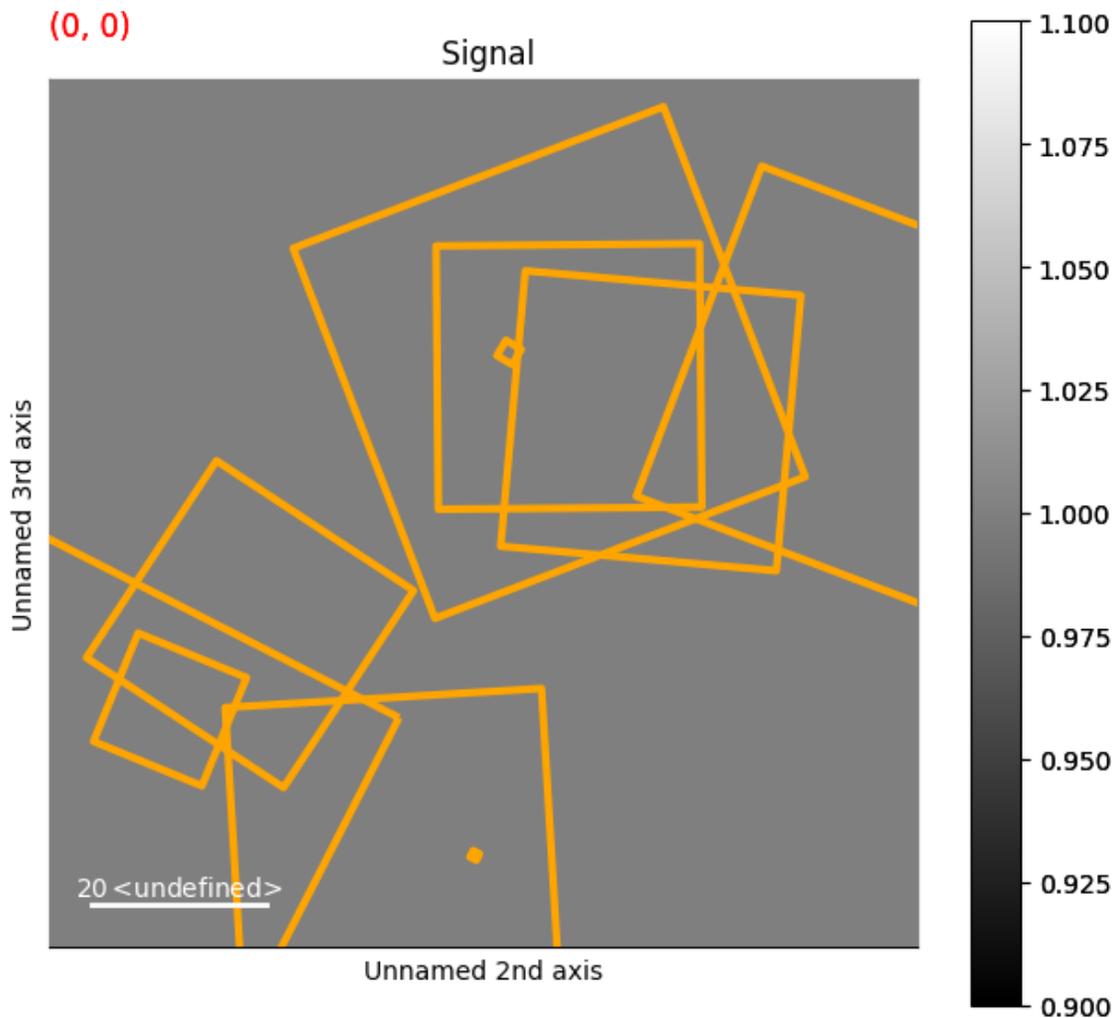
(continues on next page)

(continued from previous page)

```
m = hs.plot.markers.Squares(  
    offsets=offsets,  
    widths=widths,  
    angles=angles,  
    color="orange",  
    facecolor="none",  
    linewidth=3  
)
```

```
s2.plot()  
s2.add_marker(m)
```





sphinx_gallery_thumbnail_number = 4

Total running time of the script: (0 minutes 1.579 seconds)

22.1.15 Star Markers

Create a signal

```
import hyperspy.api as hs
import matplotlib as mpl
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

This first example shows how to draw static stars markers using the matplotlib StarPolygonCollection

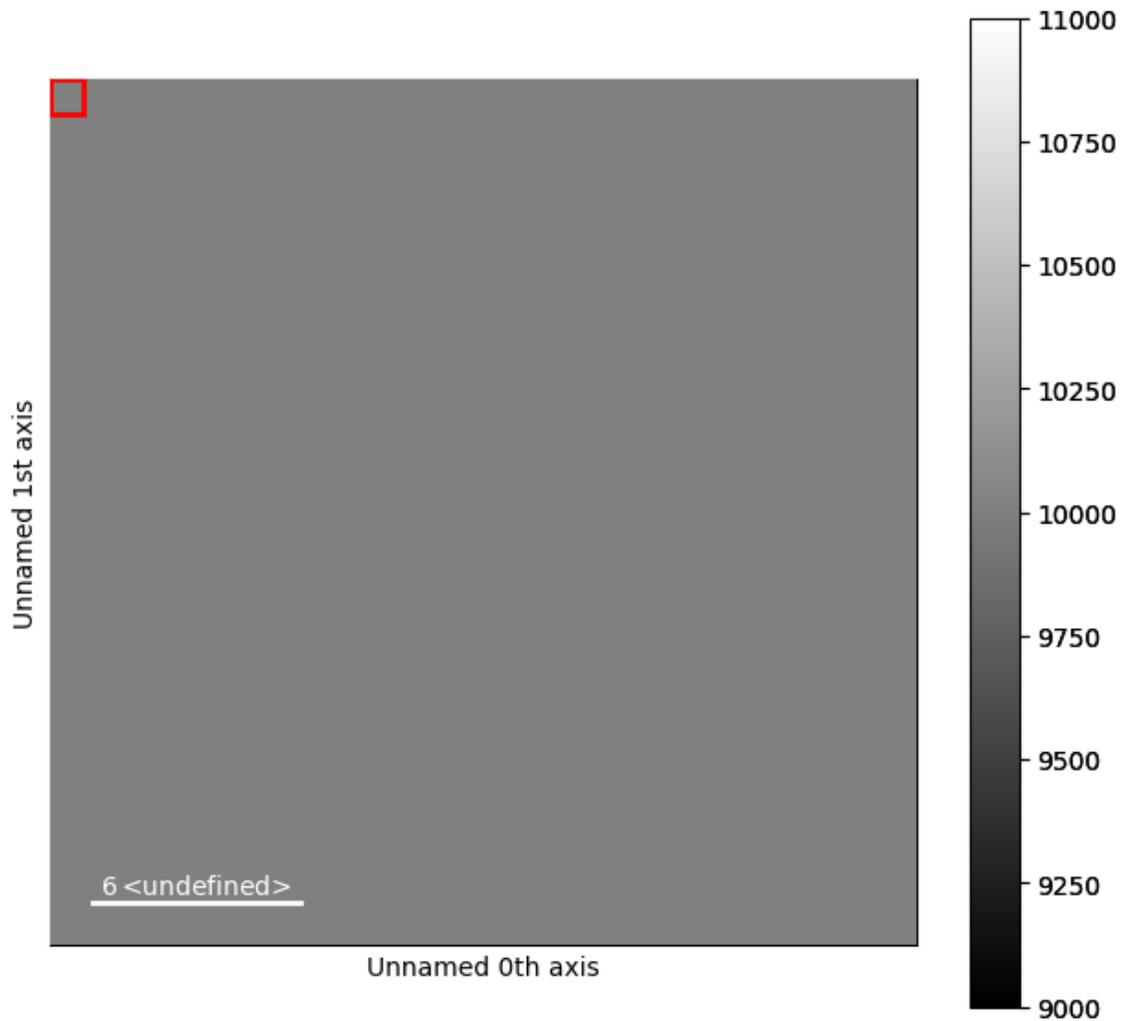
```

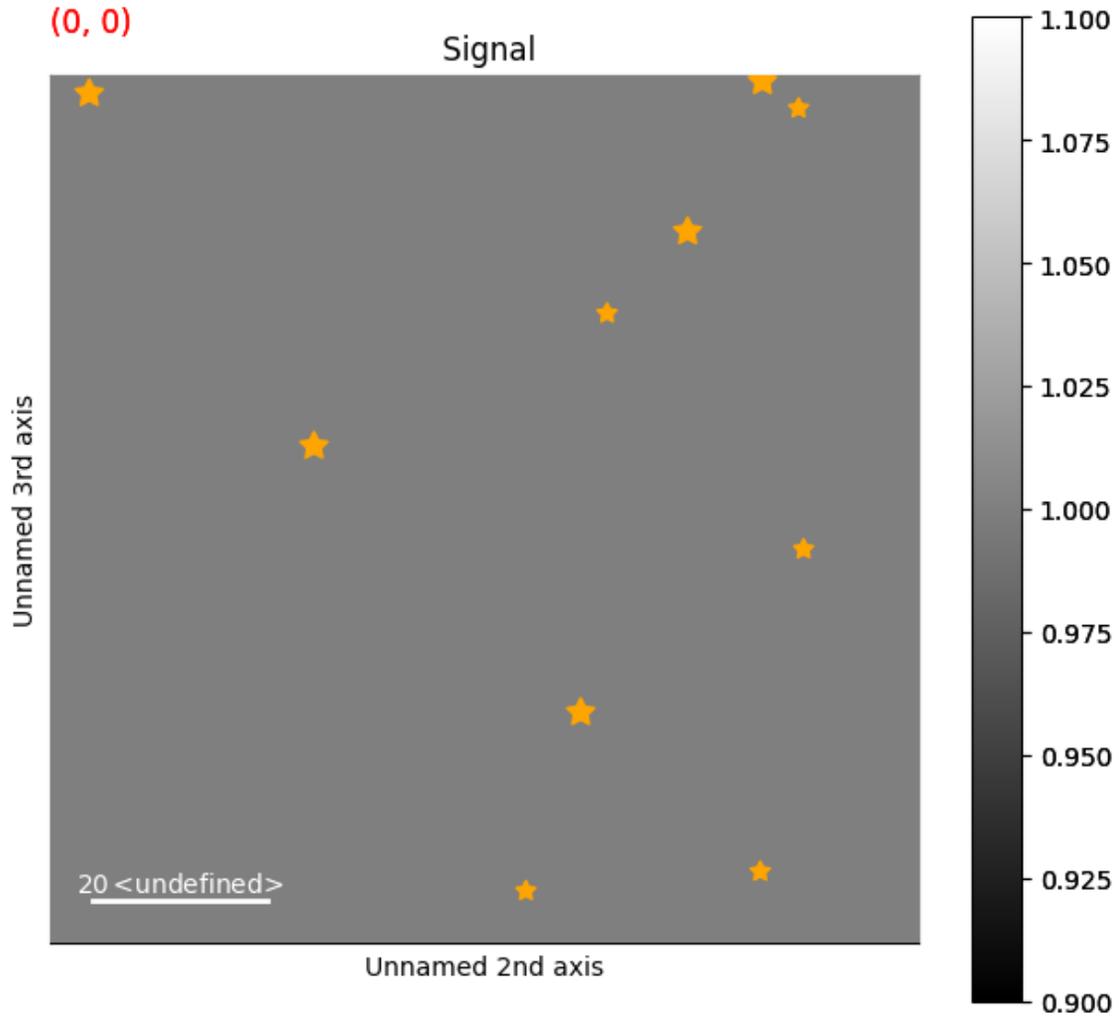
# Define the position of the boxes
offsets = rng.random((10, 2)) * 100

# every other star has a size of 50/100
m = hs.plot.markers.Markers(collection=matplotlib.collections.StarPolygonCollection,
                            offsets=offsets,
                            numsides=5,
                            color="orange",
                            sizes=(50, 100))

s.plot()
s.add_marker(m)

```





Dynamic Star Markers

This second example shows how to draw dynamic stars markers, whose position depends on the navigation coordinates

```
# Create a Signal2D with 2 navigation dimensions
s2 = hs.signals.Signal2D(data)

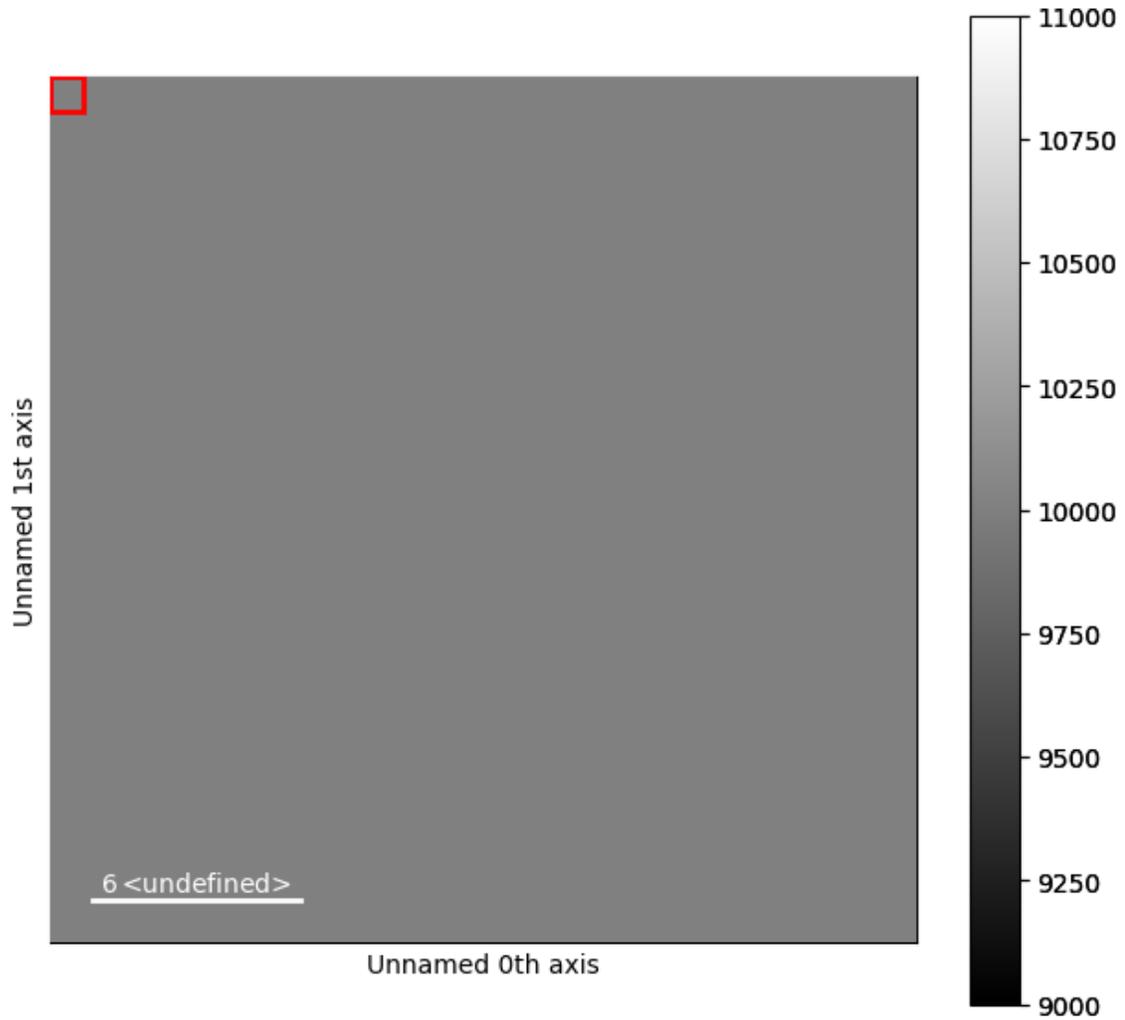
# Create a ragged array of offsets
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(offsets.shape):
    offsets[ind] = rng.random((10, 2)) * 100

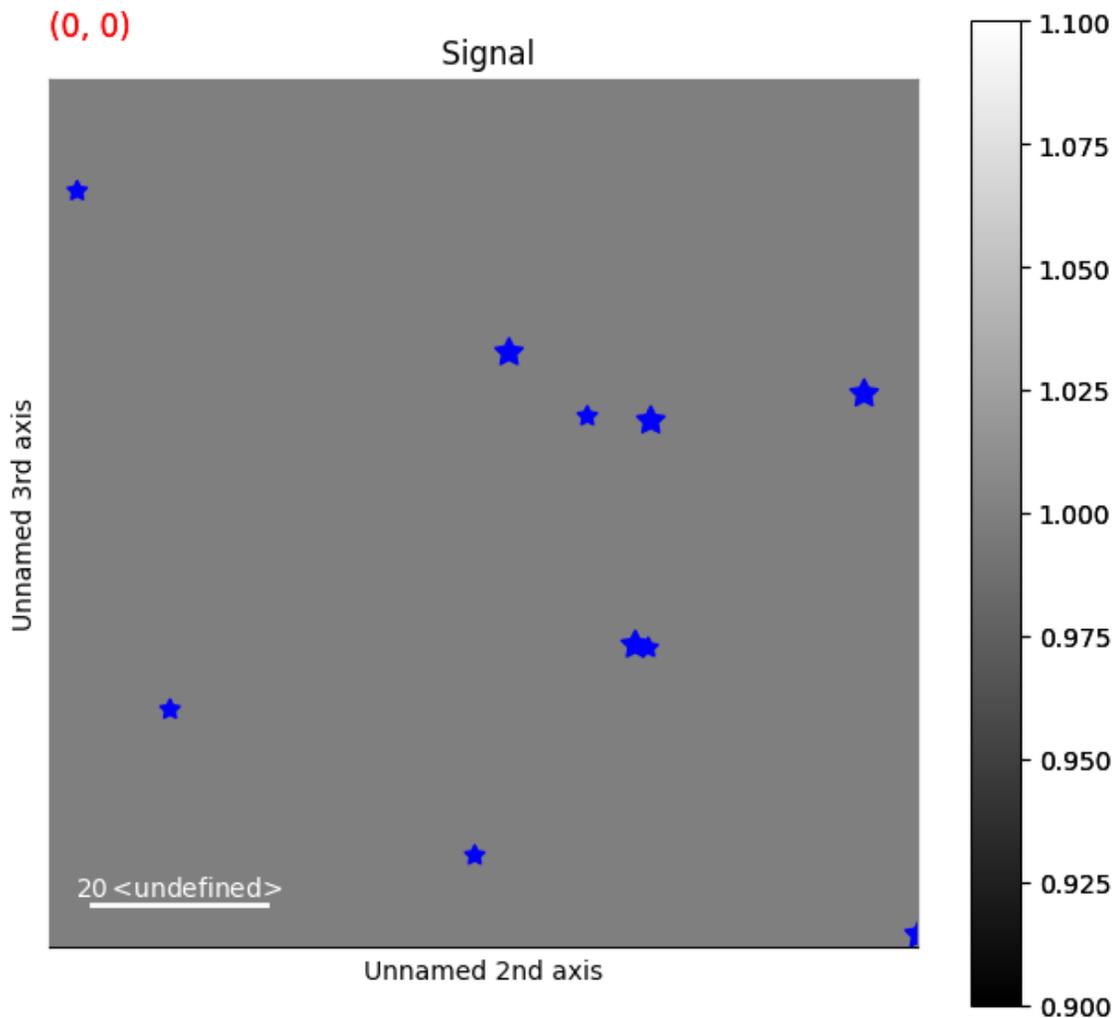
m2 = hs.plot.markers.Markers(collection=mpl.collections.StarPolygonCollection,
                             offsets=offsets,
                             numsides=5,
                             color="blue",
                             sizes=(50, 100))
```

(continues on next page)

(continued from previous page)

```
s2.plot()  
s2.add_marker(m2)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.545 seconds)

22.1.16 Rectangle Markers

Create a signal

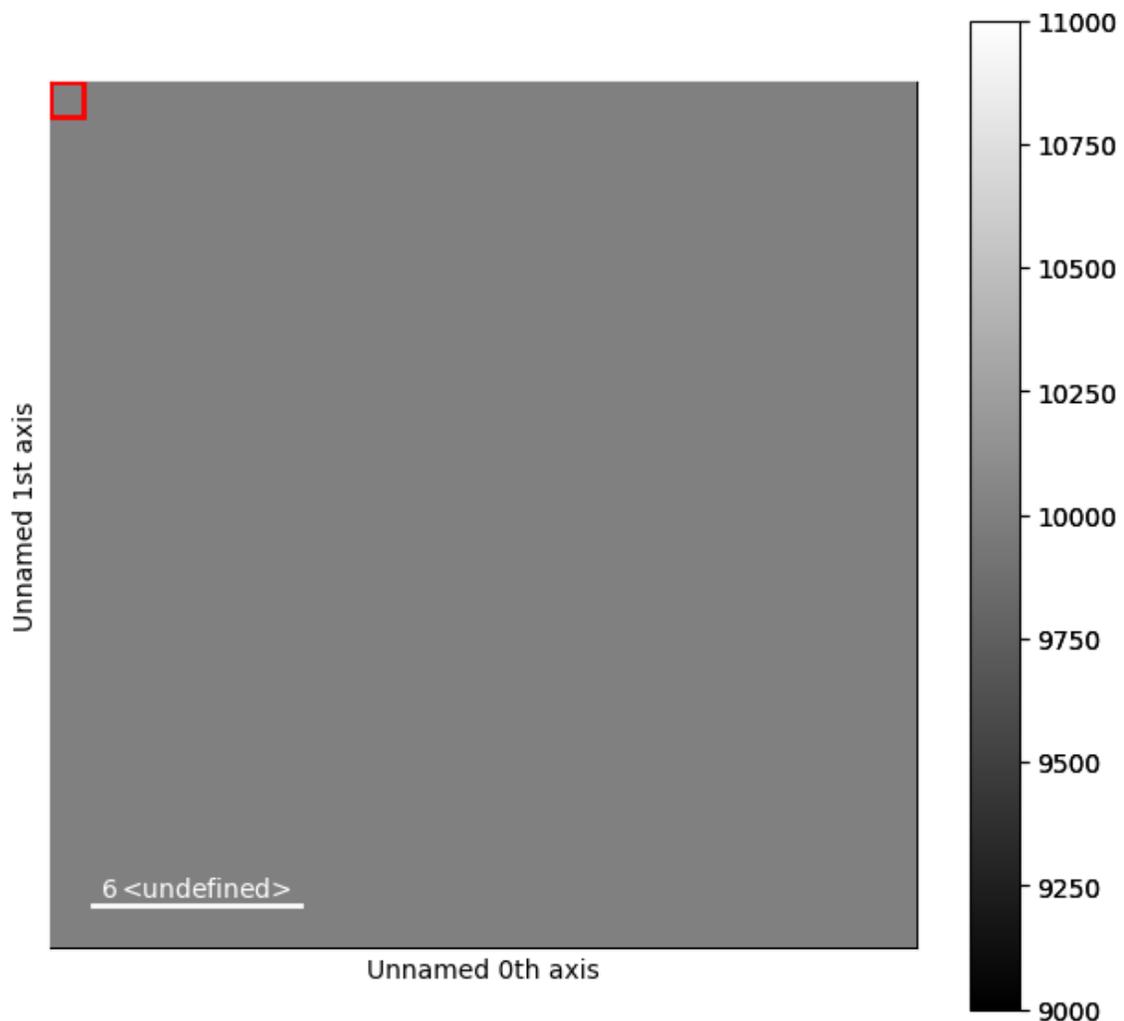
```
import hyperspy.api as hs
import numpy as np

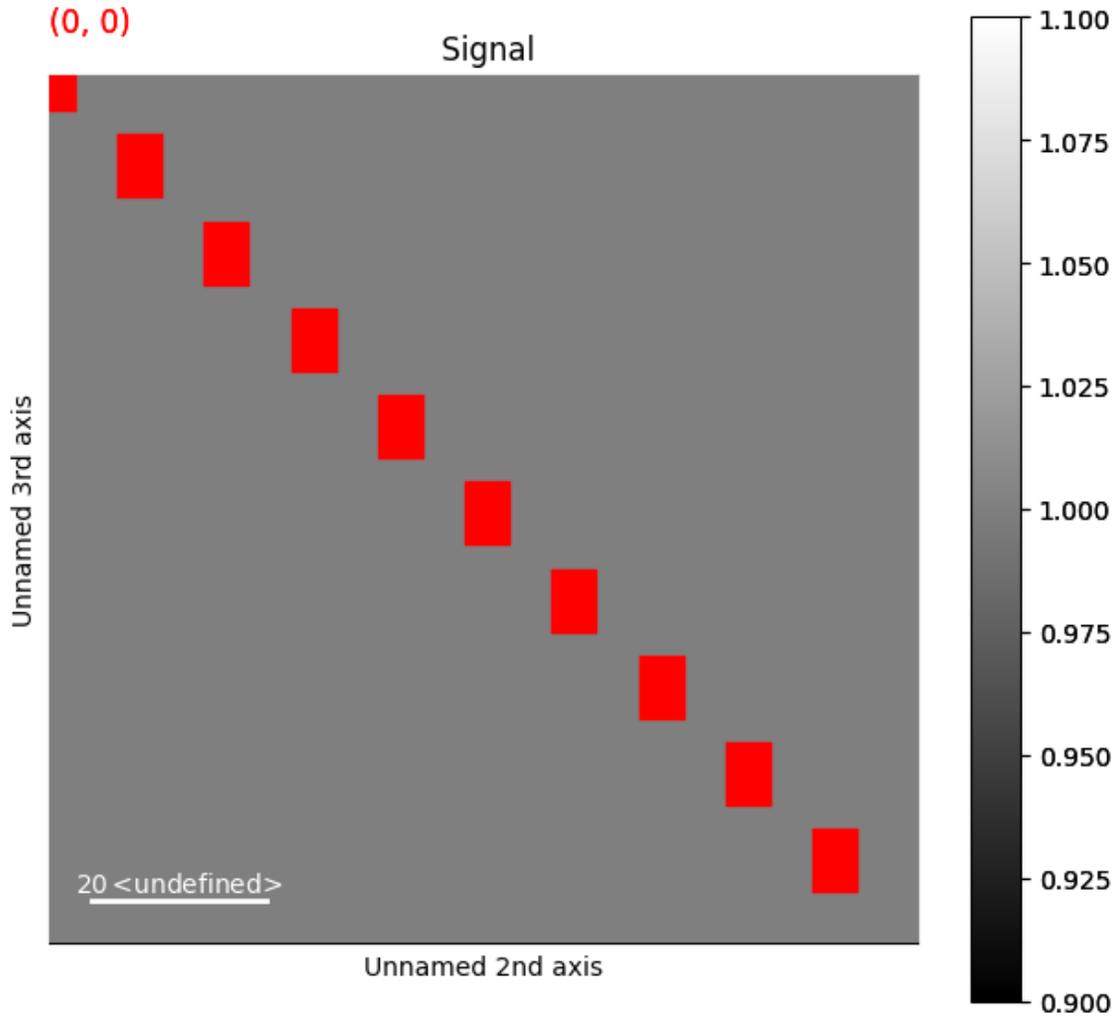
# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((25, 25, 100, 100))
s = hs.signals.Signal2D(data)
```

This first example shows how to draw static rectangle markers

```
# Define the position of the rectangles
offsets = np.array([np.arange(0, 100, 10)]*2).T

m = hs.plot.markers.Rectangles(
    offsets=offsets,
    widths=(5,),
    heights=(7,),
    angles=(0,),
    color="red",
)
s.plot()
s.add_marker(m)
```





Dynamic Rectangle Markers

This first example shows how to draw dynamic rectangle markers, whose position, widths, heights and angles depends on the navigation coordinates

```
s2 = hs.signals.Signal2D(data)

widths = np.empty(s.axes_manager.navigation_shape, dtype=object)
heights = np.empty(s.axes_manager.navigation_shape, dtype=object)
angles = np.empty(s.axes_manager.navigation_shape, dtype=object)
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)

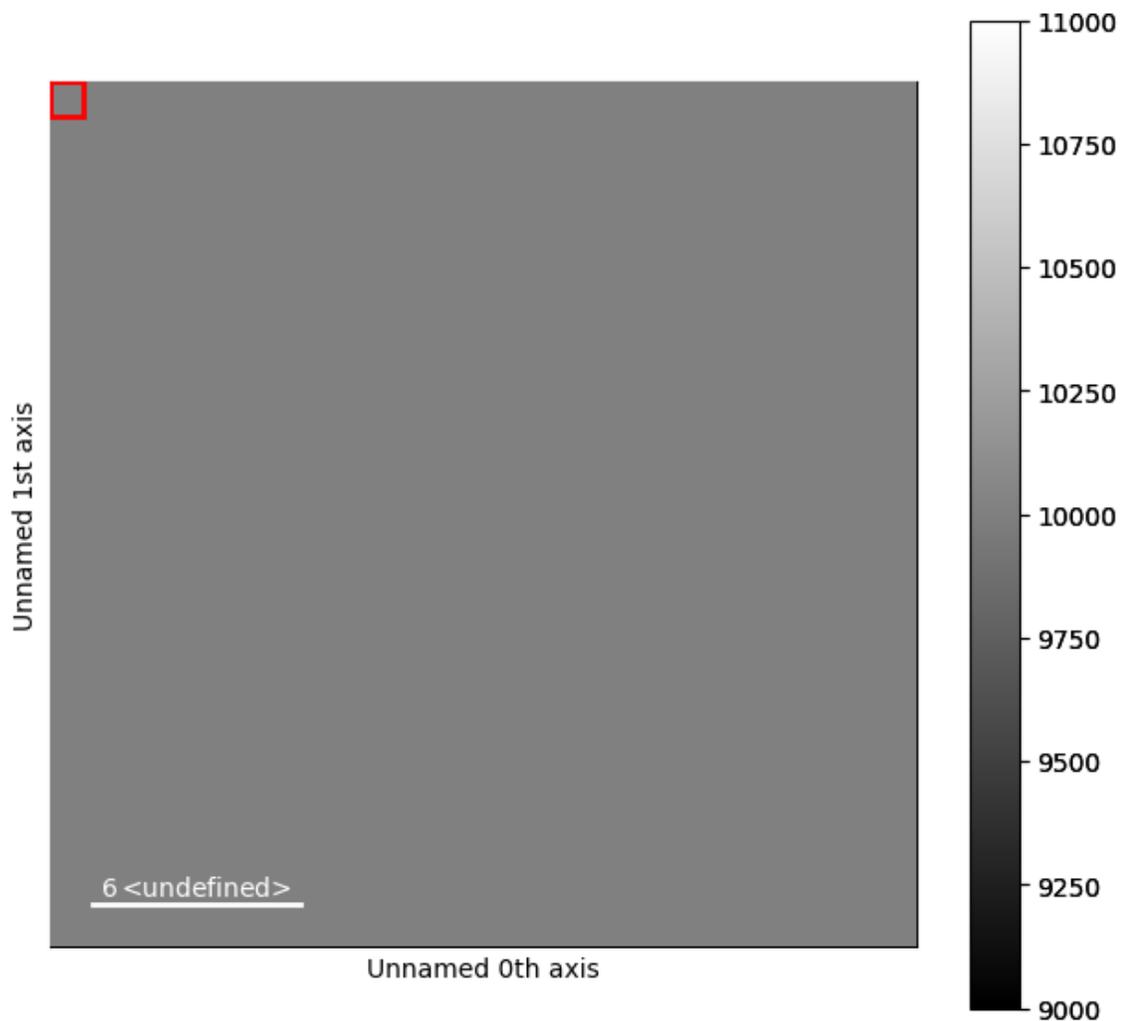
for index in np.ndindex(offsets.shape):
    widths[index] = rng.random((10, )) * 50
    heights[index] = rng.random((10, )) * 25
    angles[index] = rng.random((10, )) * 180
    offsets[index] = rng.random((10, 2)) * 100
```

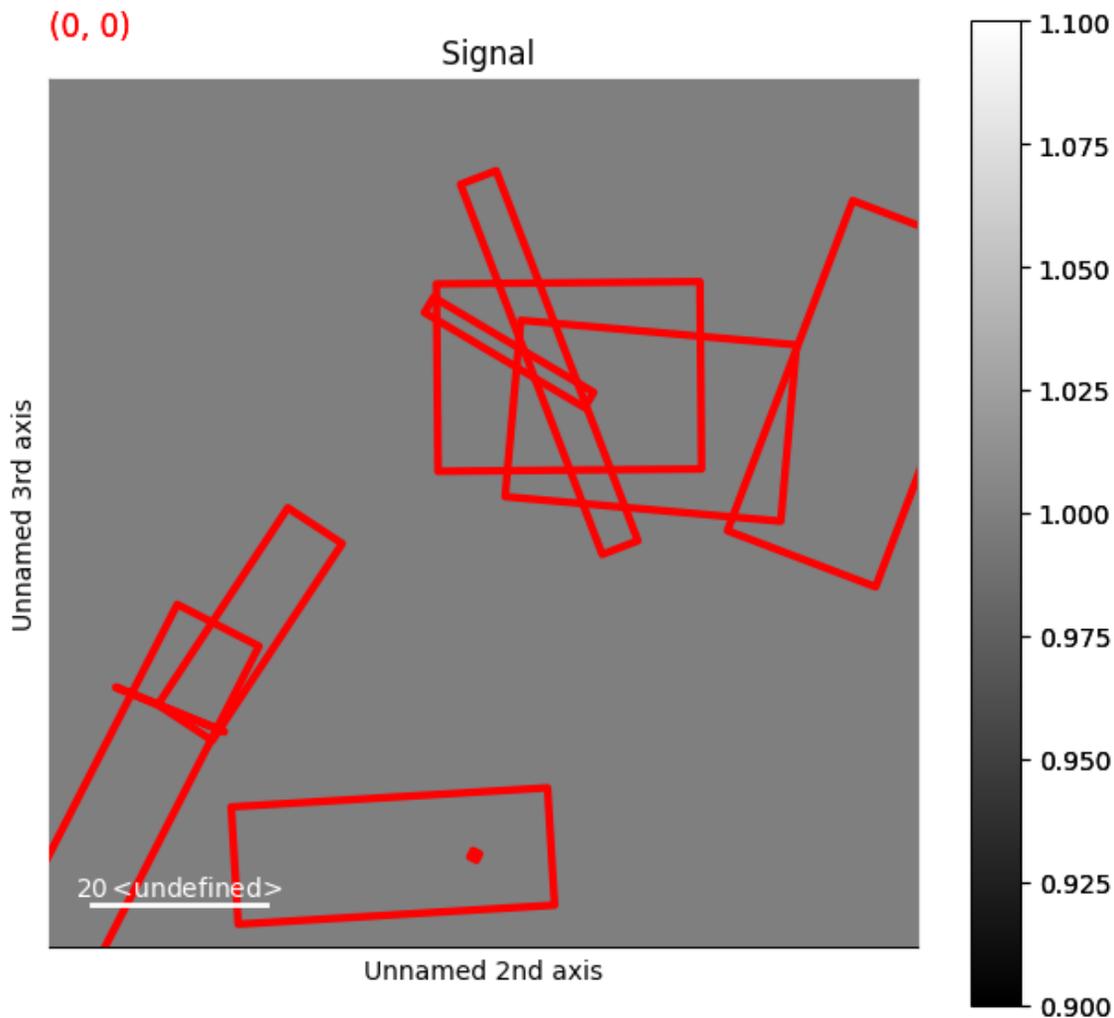
(continues on next page)

(continued from previous page)

```
m = hs.plot.markers.Rectangles(  
    offsets=offsets,  
    widths=widths,  
    heights=heights,  
    angles=angles,  
    color="red",  
    facecolor="none",  
    linewidth=3  
)
```

```
s2.plot()  
s2.add_marker(m)
```





sphinx_gallery_thumbnail_number = 4

Total running time of the script: (0 minutes 1.559 seconds)

22.1.17 Arrow markers

Create a signal

```
import hyperspy.api as hs
import numpy as np

# Create a Signal2D with 2 navigation dimensions
rng = np.random.default_rng(0)
data = np.ones((50, 100, 100))
s = hs.signals.Signal2D(data)

for axis in s.axes_manager.signal_axes:
    axis.scale = 2*np.pi / 100
```

Dynamic Arrow Markers: Changing Length

The first example shows how to change the length of the arrows when changing the navigation coordinates

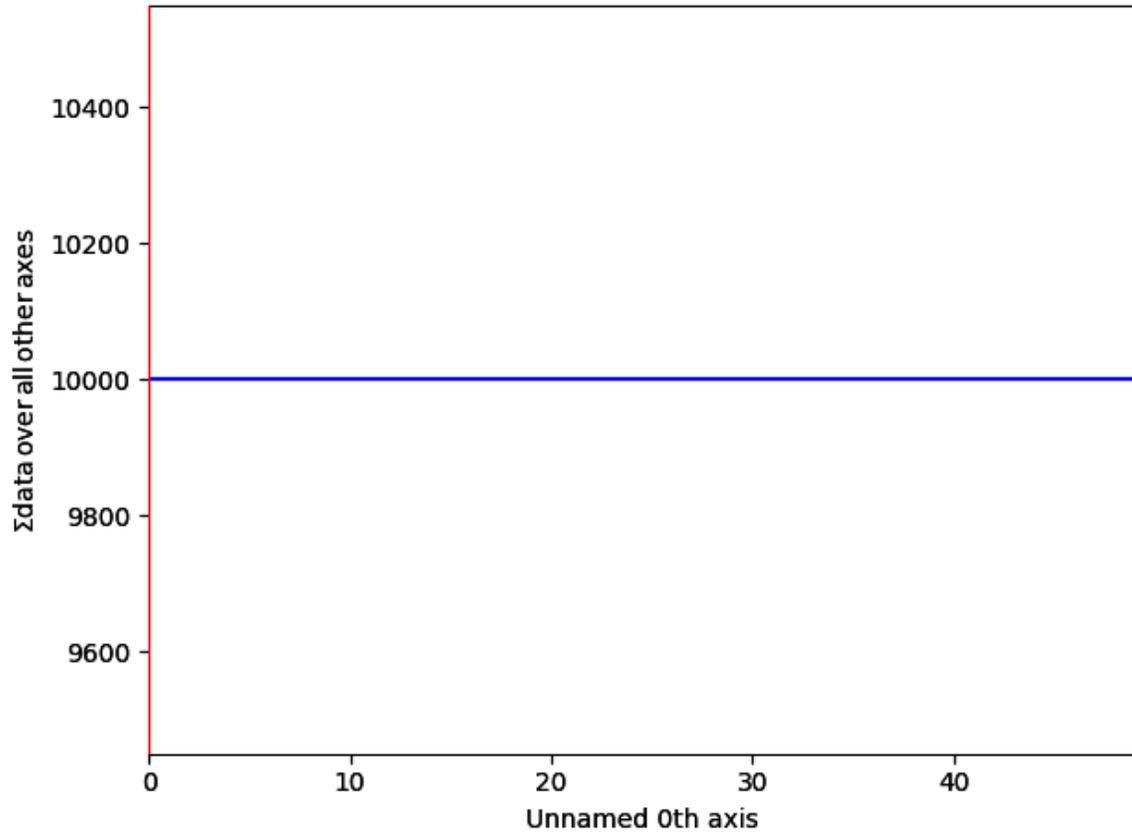
```
X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
offsets = np.column_stack((X.ravel(), Y.ravel()))

weight = np.cos(np.linspace(0, 4*np.pi, num=50))

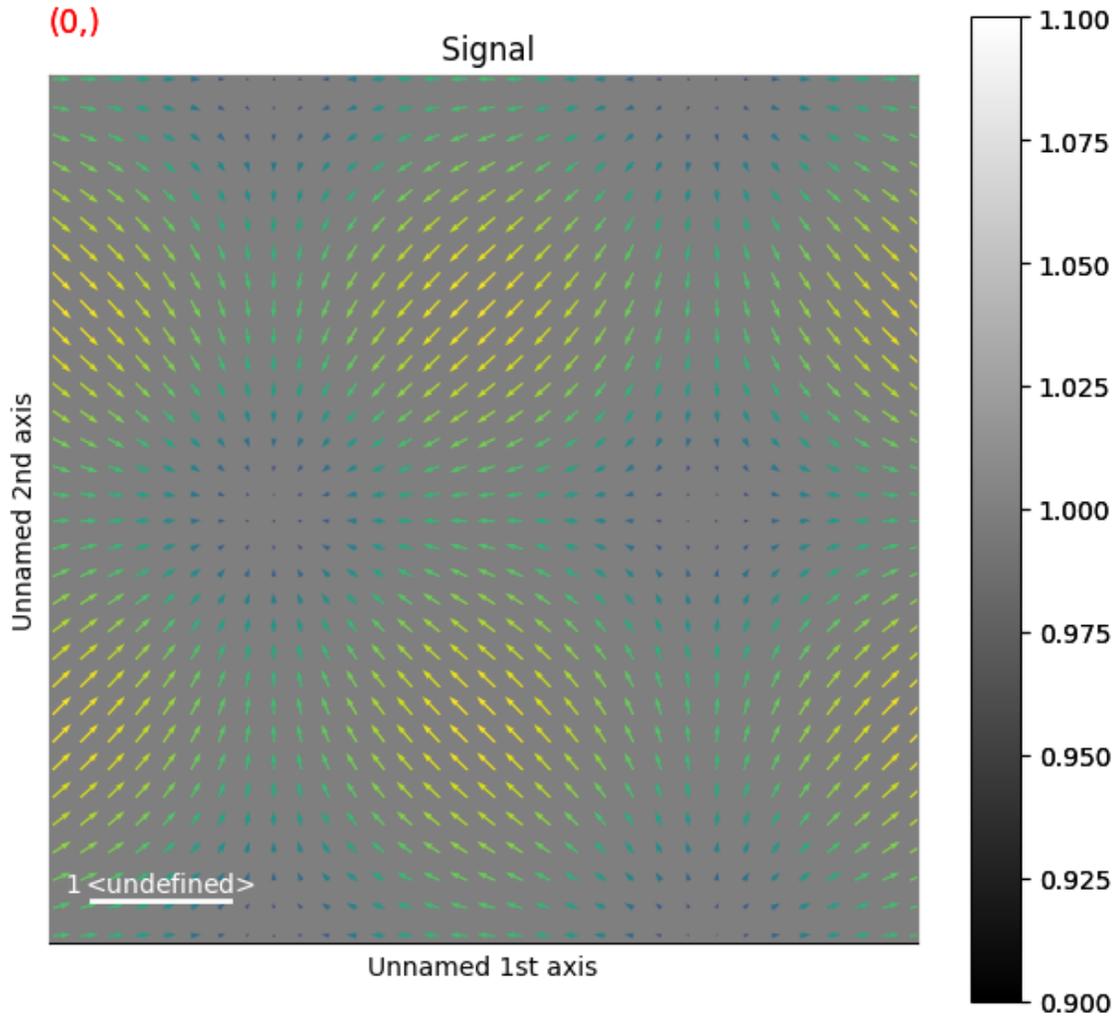
U = np.empty(s.axes_manager.navigation_shape, dtype=object)
V = np.empty(s.axes_manager.navigation_shape, dtype=object)
C = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(U.shape):
    U[ind] = np.cos(X).ravel() / 7.5 * weight[ind]
    V[ind] = np.sin(Y).ravel() / 7.5 * weight[ind]
    C[ind] = np.hypot(U[ind], V[ind])

m = hs.plot.markers.Arrows(
    offsets,
    U,
    V,
    C=C
)

s.plot()
s.add_marker(m)
```



.



Dynamic Arrow Markers: Changing Position

The second example shows how to change the position of the arrows when changing the navigation coordinates

```
X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
U = np.cos(X).ravel() / 7.5
V = np.sin(Y).ravel() / 7.5
C = np.hypot(U, V)

weight_x = np.sin(np.linspace(0, 2*np.pi, num=50))
weight_y = np.cos(np.linspace(0, 2*np.pi, num=50))

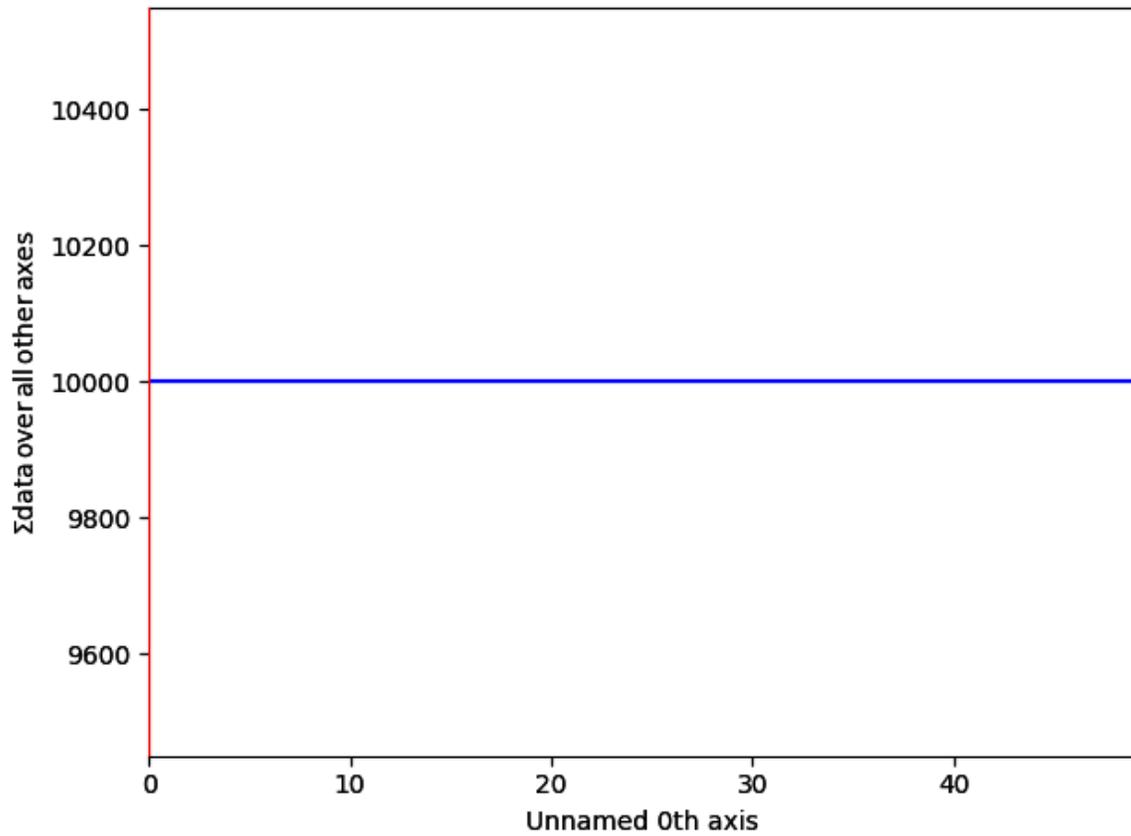
offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
for ind in np.ndindex(offsets.shape):
    offsets[ind] = np.column_stack((X.ravel() + weight_x[ind], Y.ravel() + weight_
    ↪y[ind]))

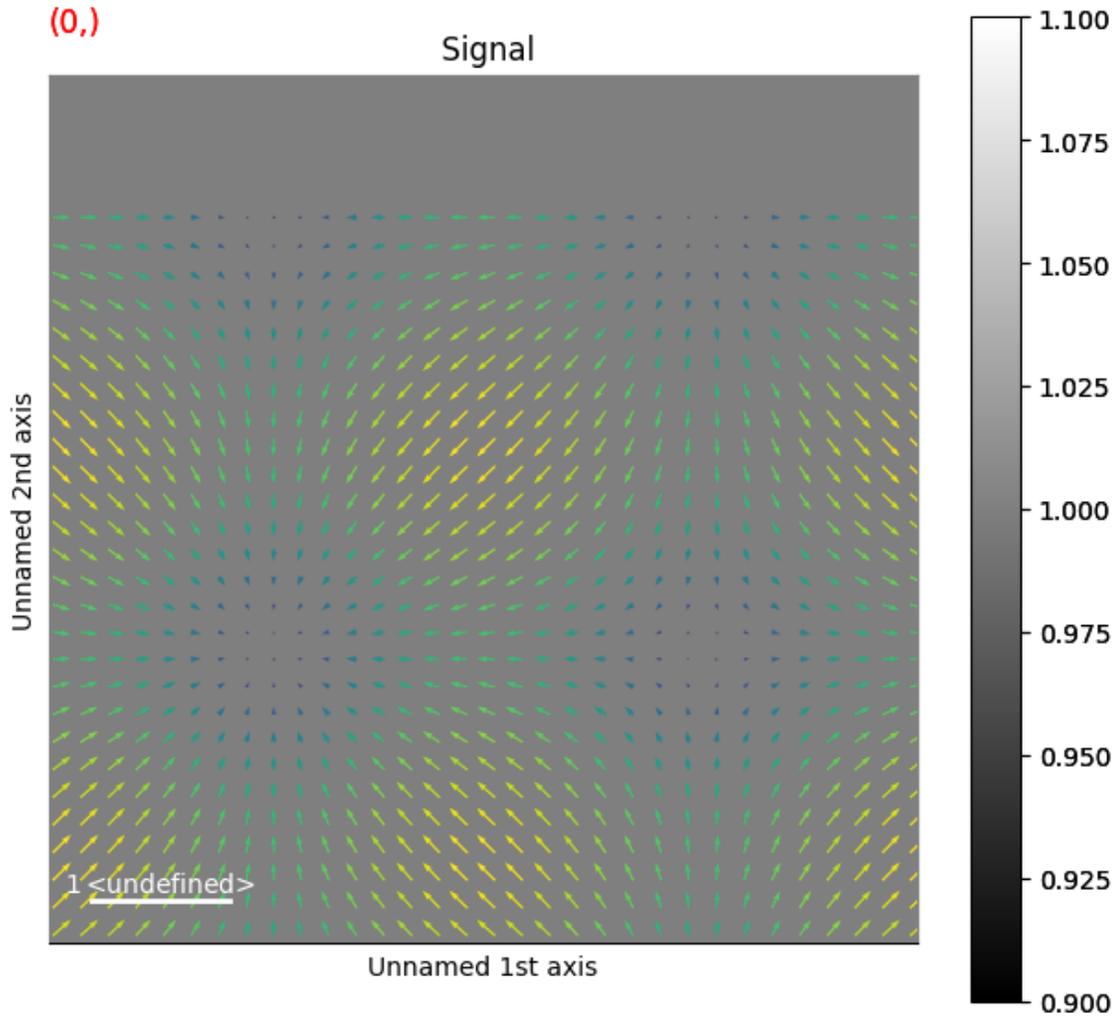
m = hs.plot.markers.Arrows(
```

(continues on next page)

(continued from previous page)

```
offsets,  
U,  
V,  
C=C  
)  
  
s.plot()  
s.add_marker(m)
```



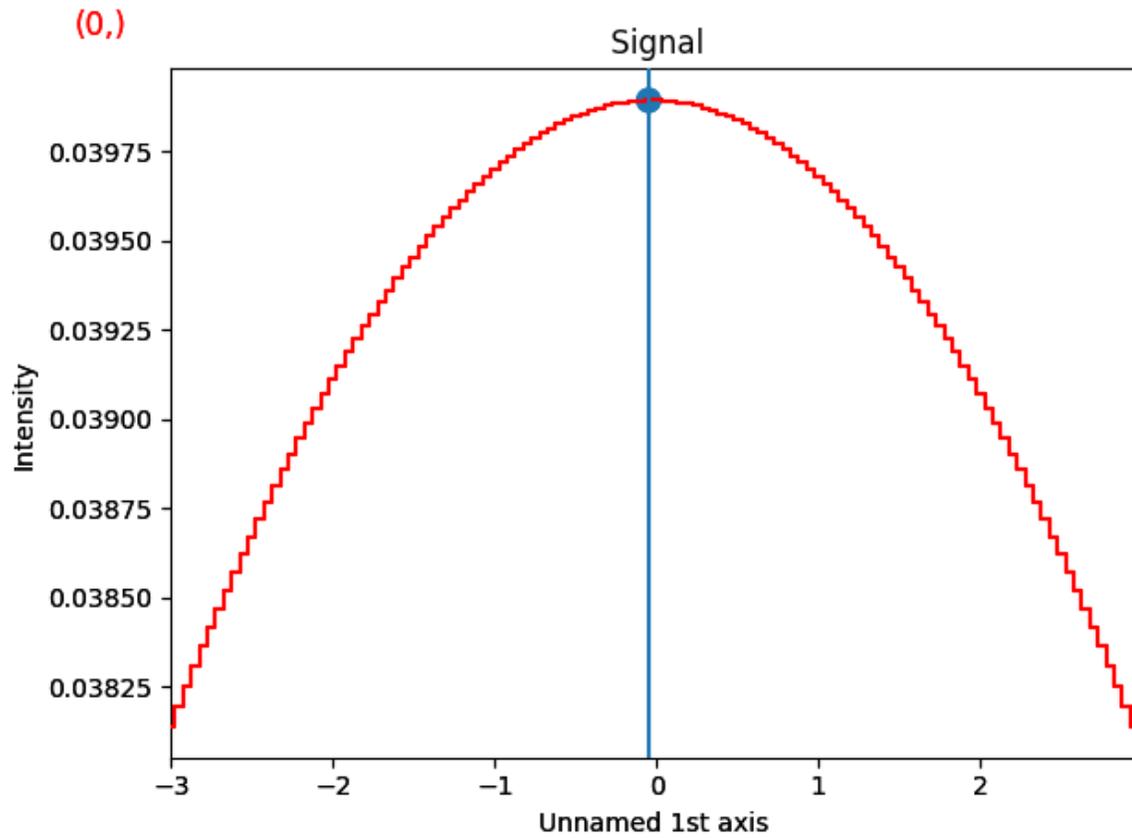
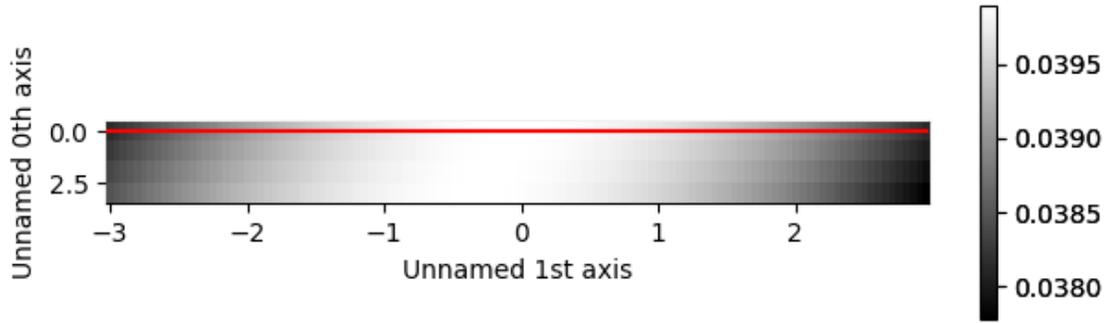


sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 1.773 seconds)

22.1.18 Creating Markers from a signal

This example shows how to create markers from a signal. This is useful for creating lazy markers from some operation such as peak finding on a signal. Here we show how to create markers from a simple map function which finds the maximum value and plots a marker at that position.



```
[
                                ] | 0% Completed | 154.43 us[-0.05]
[0.0398941]
[-0.1]
[0.03989411]
[-0.2]
[0.03989412]
[-0.3]
[0.03989413]

[#####] | 100% Completed | 100.55 ms

[
                                ] | 0% Completed | 155.49 us
```

(continues on next page)

(continued from previous page)

[#####] | 100% Completed | 100.40 ms

```

import numpy as np
import hyperspy.api as hs

# Making some artificial data
def find_maxima(data, scale, offset):
    ind = np.array(np.unravel_index(np.argmax(data, axis=None), data.shape)).astype(int)
    d = data[ind]
    ind = ind * scale + offset # convert to physical units
    print(ind)
    print(d)
    return np.array(
        [
            [ind[0], d[0]],
        ]
    )

def find_maxima_lines(data, scale, offset):
    ind = np.array(np.unravel_index(np.argmax(data, axis=None), data.shape)).astype(int)
    ind = ind * scale + offset # convert to physical units
    return ind

def gaussian(x, mu, sig):
    return (
        1.0 / (np.sqrt(2.0 * np.pi) * sig) * np.exp(-np.power((x - mu) / sig, 2.0) / 2)
    )

data = np.empty((4, 120))
for i in range(4):
    x_values = np.linspace(-3 + i * 0.1, 3 + i * 0.1, 120)
    data[i] = gaussian(x_values, mu=0, sig=10)

s = hs.signals.Signal1D(data)
s.axes_manager.signal_axes[0].scale = 6 / 120
s.axes_manager.signal_axes[0].offset = -3

scale = s.axes_manager.signal_axes[0].scale
offset = s.axes_manager.signal_axes[0].offset
max_values = s.map(find_maxima, scale=scale, offset=offset, inplace=False, ragged=True)
max_values_lines = s.map(
    find_maxima_lines, scale=scale, offset=offset, inplace=False, ragged=True

```

(continues on next page)

(continued from previous page)

```

)
point_markers = hs.plot.markers.Points.from_signal(max_values, signal_axes=None)
line_markers = hs.plot.markers.VerticalLines.from_signal(
    max_values_lines, signal_axes=None
)

s.plot()
s.add_marker(point_markers)
s.add_marker(line_markers)

```

Total running time of the script: (0 minutes 0.919 seconds)

22.1.19 Transforms and Units

This example shows how to use both the `offset_transform` and `transforms` parameters for markers

Create a signal

```

import hyperspy.api as hs
import numpy as np

rng = np.random.default_rng()
data = np.arange(1, 101).reshape(10, 10)*2 + rng.random((10, 10))
signal = hs.signals.Signal1D(data)

```

The first example shows how to draw markers which are relative to some 1D signal. This is how the EDS and EELS Lines are implemented in the `exspy` package.

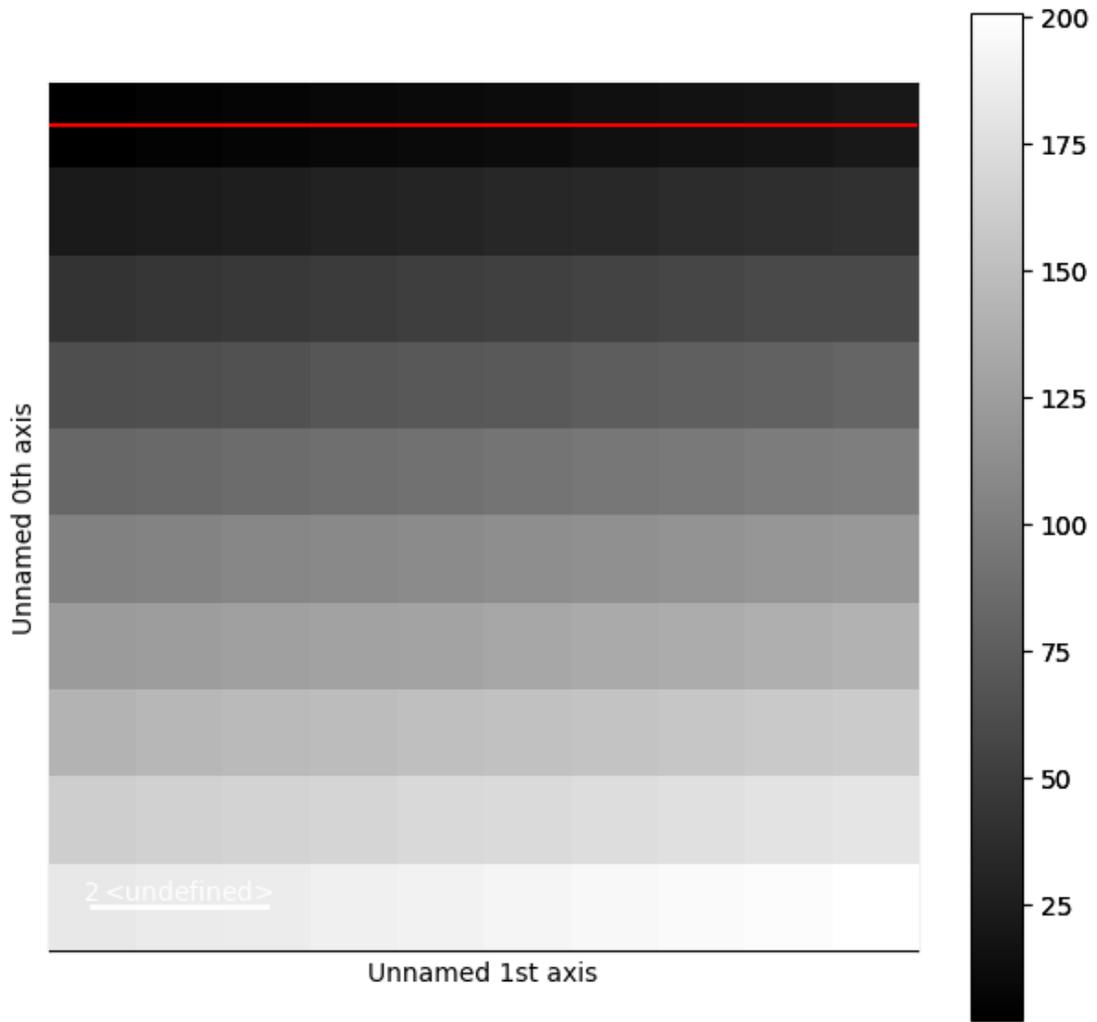
```

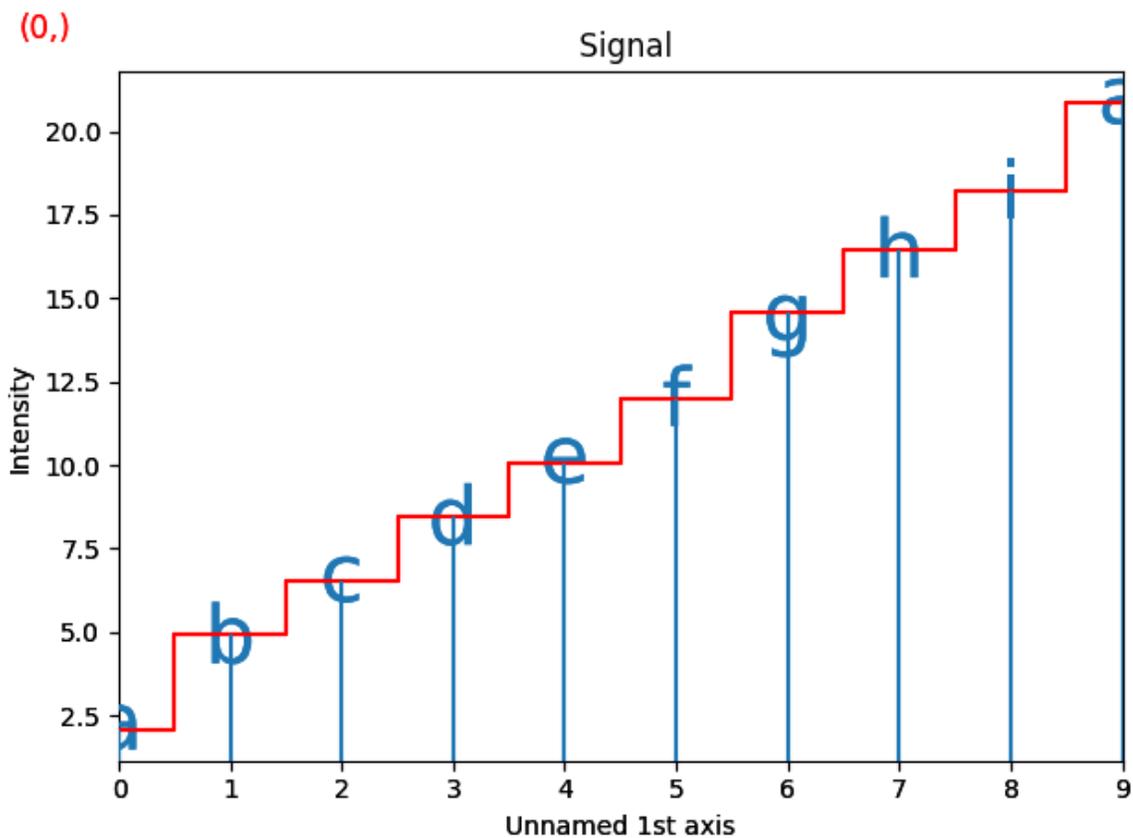
segments = np.zeros((10, 2, 2)) # line segments for relative markers
segments[:, 1, 1] = 1 # set y values end (1 means to the signal curve)
segments[:, 0, 0] = np.arange(10).reshape(10) # set x for line start
segments[:, 1, 0] = np.arange(10).reshape(10) # set x for line stop

offsets = np.zeros((10,2)) # offsets for texts positions
offsets[:, 1] = 1 # set y value for text position ((1 means to the signal curve))
offsets[:, 0] = np.arange(10).reshape(10) # set x for line start

markers = hs.plot.markers.Lines(segments=segments, transform="relative")
texts = hs.plot.markers.Texts(offsets=offsets,
                              texts=["a", "b", "c", "d", "e", "f", "g", "h", "i"],
                              sizes=10,
                              offset_transform="relative",
                              shift=0.005) # shift in axes units for some constant
↪ displacement
signal.plot()
signal.add_marker(markers)
signal.add_marker(texts)

```

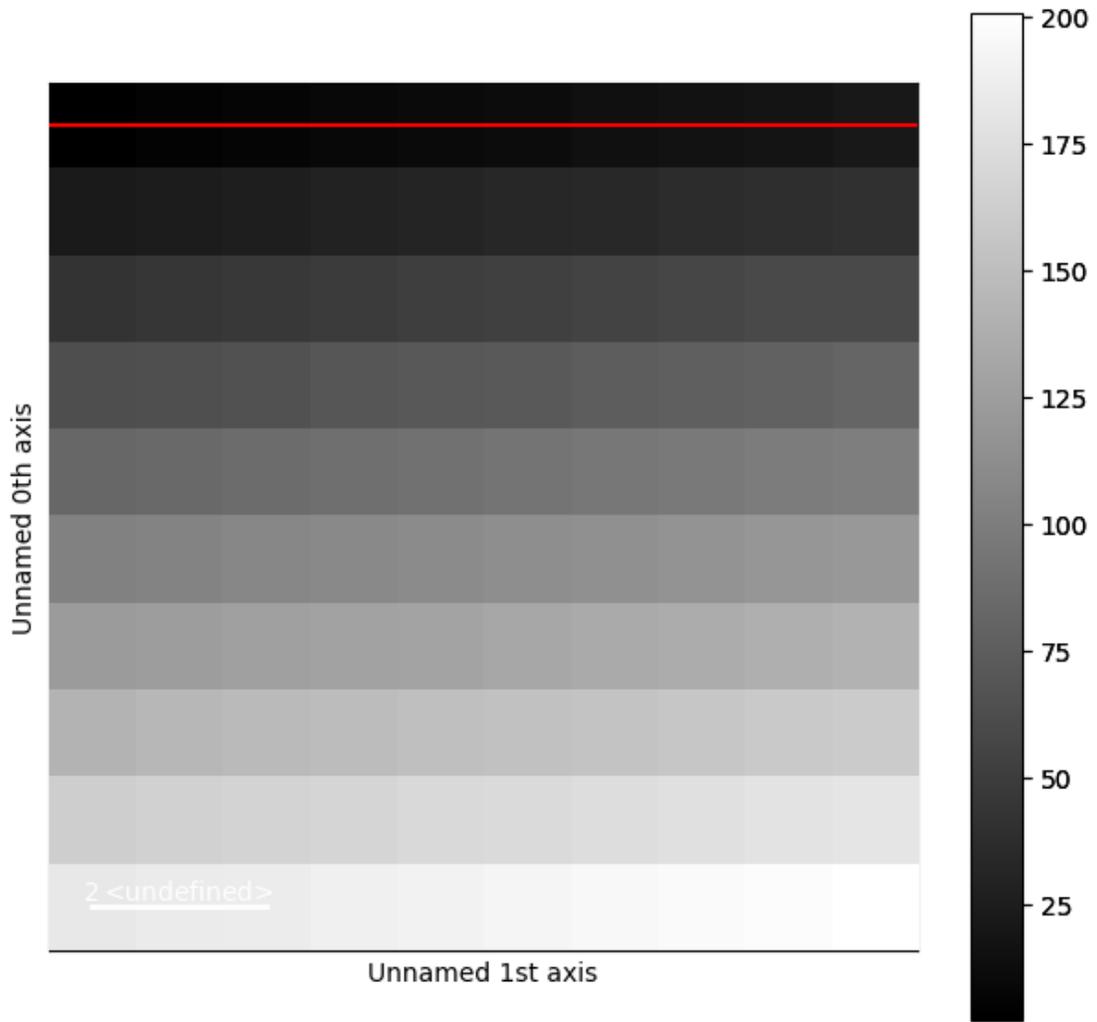


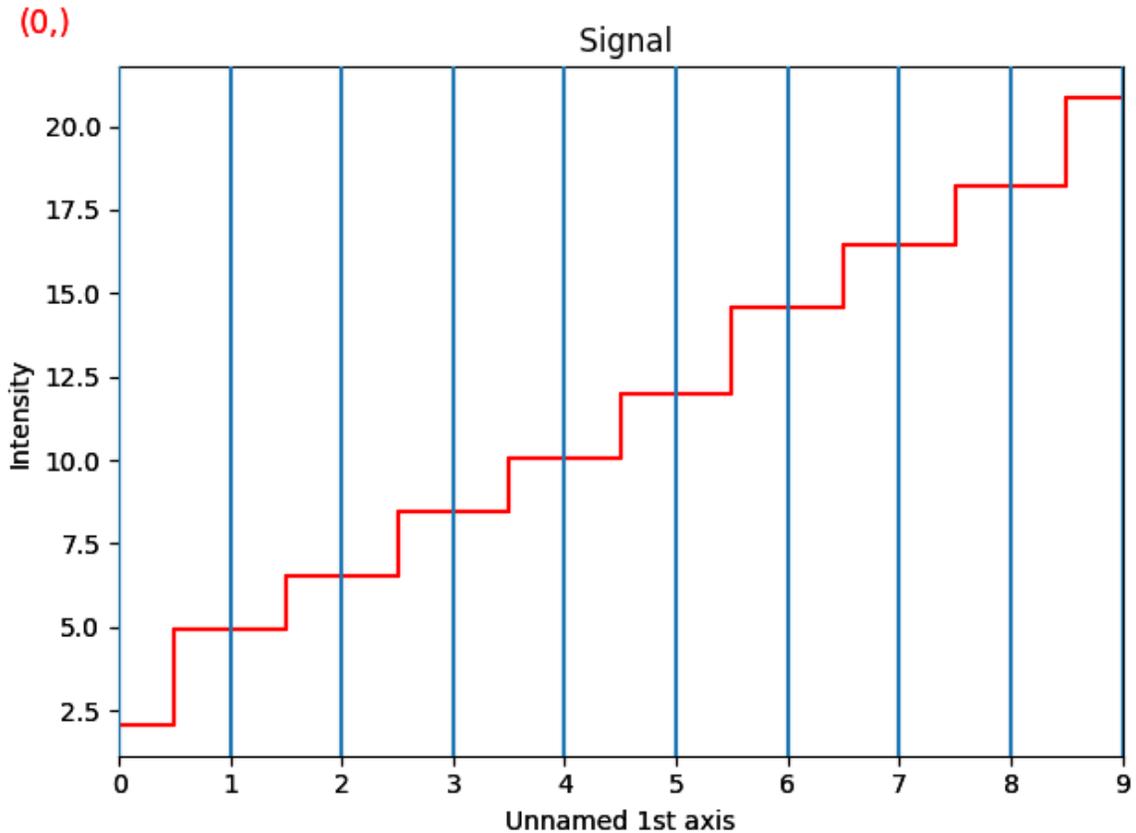


The second example shows how to draw markers which extend to the edges of the axes. This is how the VerticalLines and HorizontalLines markers are implemented.

```
markers = hs.plot.markers.Lines(segments=segments,
                                transform="xaxis")
```

```
signal.plot()
signal.add_marker(markers)
```





The third example shows how an `offset_transform` of 'axes' can be used to annotate a signal.

The size of the marker is specified in units defined by the transform, in this case "axis_scale", "yaxis_scale" or "display"

```
offsets = [[1, 13.5], ] # offsets for positions
sizes = 1
units = 'x'
offset_transform = 'data'
string = (f"          sizes={sizes}, offset_transform='{offset_transform}', units='{units}',
↪ offsets={offsets}")

marker1text = hs.plot.markers.Texts(offsets=offsets,
                                   texts=string,
                                   sizes=1,
                                   horizontalalignment="left",
                                   verticalalignment="baseline",
                                   offset_transform=offset_transform)

marker = hs.plot.markers.Points(offsets=offsets,
                                sizes=sizes, units=units, offset_transform=offset_transform)

offsets = [[.1, .1], ] # offsets for positions
sizes = 10
```

(continues on next page)

(continued from previous page)

```

units = 'points'
offset_transform = 'axes'
string = (f" sizes={sizes}, offset_transform='{offset_transform}', units='{units}',
↪ offsets={offsets}"),)

marker2text = hs.plot.markers.Texts(offsets=offsets,
                                   texts=string,
                                   sizes=1,
                                   horizontalalignment="left",
                                   verticalalignment="baseline",
                                   offset_transform=offset_transform)

marker2 = hs.plot.markers.Points(offsets=offsets,
                                sizes=sizes, units=units, offset_transform=offset_transform)

offsets = [[.1, .8], ] # offsets for positions
sizes = 1
units = 'y'
offset_transform = 'axes'
string = (f" sizes={sizes}, offset_transform='{offset_transform}', units='{units}',
↪ offsets={offsets}"),)

marker3text = hs.plot.markers.Texts(offsets=offsets,
                                   texts=string,
                                   sizes=1,
                                   horizontalalignment="left",
                                   verticalalignment="baseline",
                                   offset_transform=offset_transform)

marker3 = hs.plot.markers.Points(offsets=offsets,
                                sizes=sizes, units=units, offset_transform=offset_transform)

offsets = [[1, 7.5], ] # offsets for positions
sizes = 1
units = 'xy'
offset_transform = 'data'
string = (f" sizes={sizes}, offset_transform='{offset_transform}', units='{units}',
↪ offsets={offsets}"),)

marker4text = hs.plot.markers.Texts(offsets=offsets,
                                   texts=string,
                                   sizes=1,
                                   horizontalalignment="left",
                                   verticalalignment="baseline",
                                   offset_transform=offset_transform)

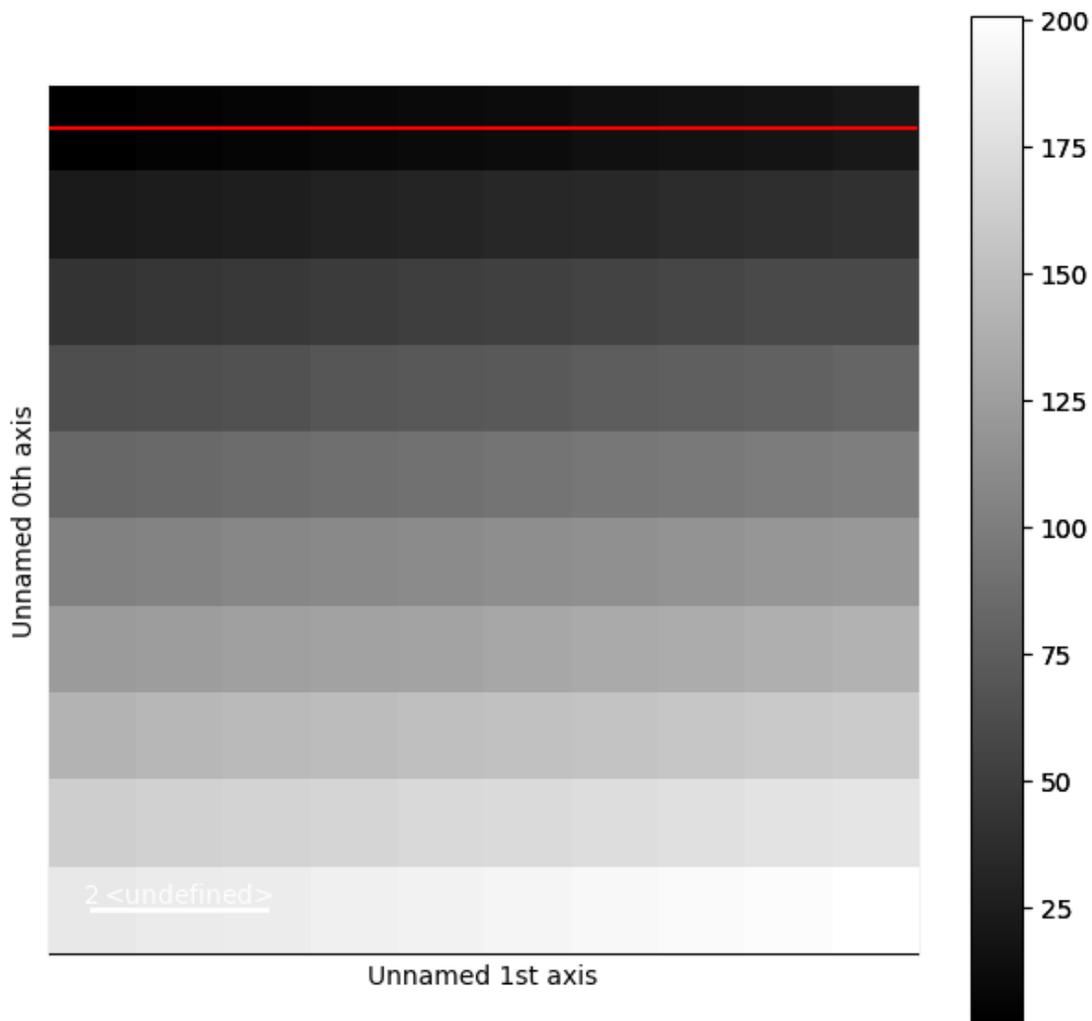
marker4 = hs.plot.markers.Points(offsets=offsets,
                                sizes=sizes, units=units, offset_transform=offset_transform)

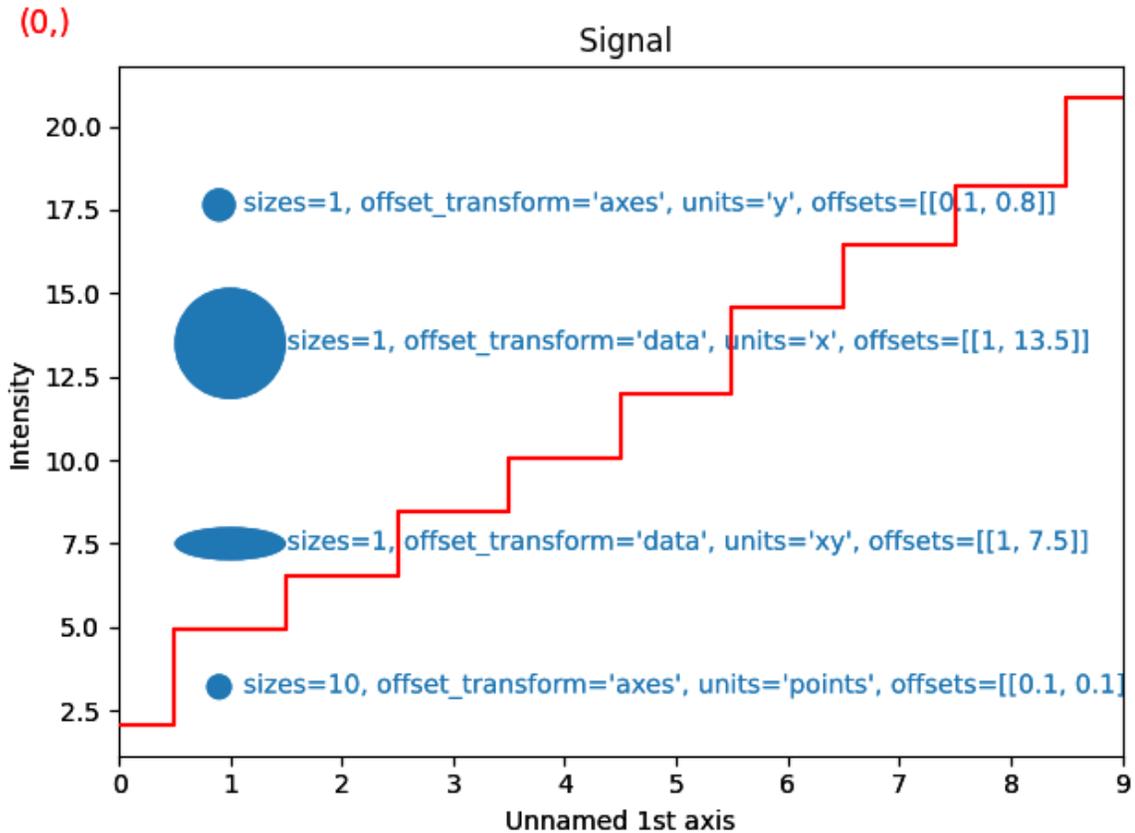
```

(continues on next page)

(continued from previous page)

```
signal.plot()  
signal.add_marker(marker)  
signal.add_marker(marker1text)  
signal.add_marker(marker2)  
signal.add_marker(marker2text)  
signal.add_marker(marker3)  
signal.add_marker(marker3text)  
signal.add_marker(marker4)  
signal.add_marker(marker4text)
```





sphinx_gallery_thumbnail_number = 2

Total running time of the script: (0 minutes 2.503 seconds)

22.2 Signal Creation

Below is a gallery of examples on creating a signal and plotting.

22.2.1 Creates a signal1D from a text file

This example creates a signal from tabular data imported from a txt file using `numpy.loadtxt()`. The signal axis and the EELS intensity values are given by the first and second columns, respectively.

The tabular data are taken from <https://eelsdb.eu/spectra/la2nio4-structure-of-k2nif4/>

```
import numpy as np
import hyperspy.api as hs
```

Read tabular data from a text file:

```
x, y = np.loadtxt("La2NiO4_eels.txt", unpack=True)
```

Define the axes of the signal and then create the signal:

```
axes = [  
    # use values from first column to define non-uniform signal axis  
    dict(axis=x, name="Energy", units="eV"),  
]  
  
s = hs.signals.Signal1D(y, axes=axes)
```

Convert the non-uniform axis to a uniform axis, because non-uniform axes do not support all functionalities of HyperSpy. In this case, the error introduced during conversion to uniform *scale* is negligible.

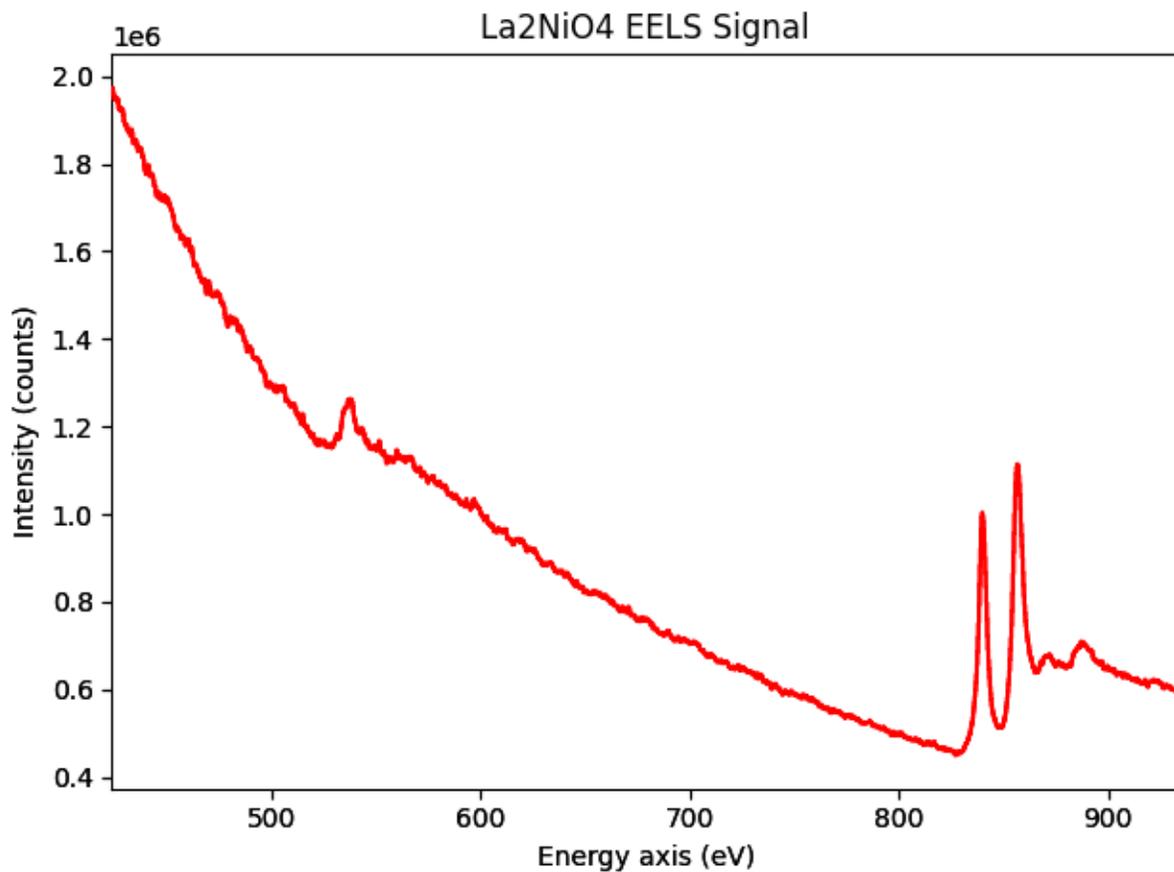
```
s.axes_manager.signal_axes[0].convert_to_uniform_axis()
```

Set title of the dataset and label for the data axis:

```
s.metadata.set_item("General.title", "La2NiO4 EELS")  
s.metadata.set_item("Signal.quantity", "Intensity (counts)")
```

Plot the dataset:

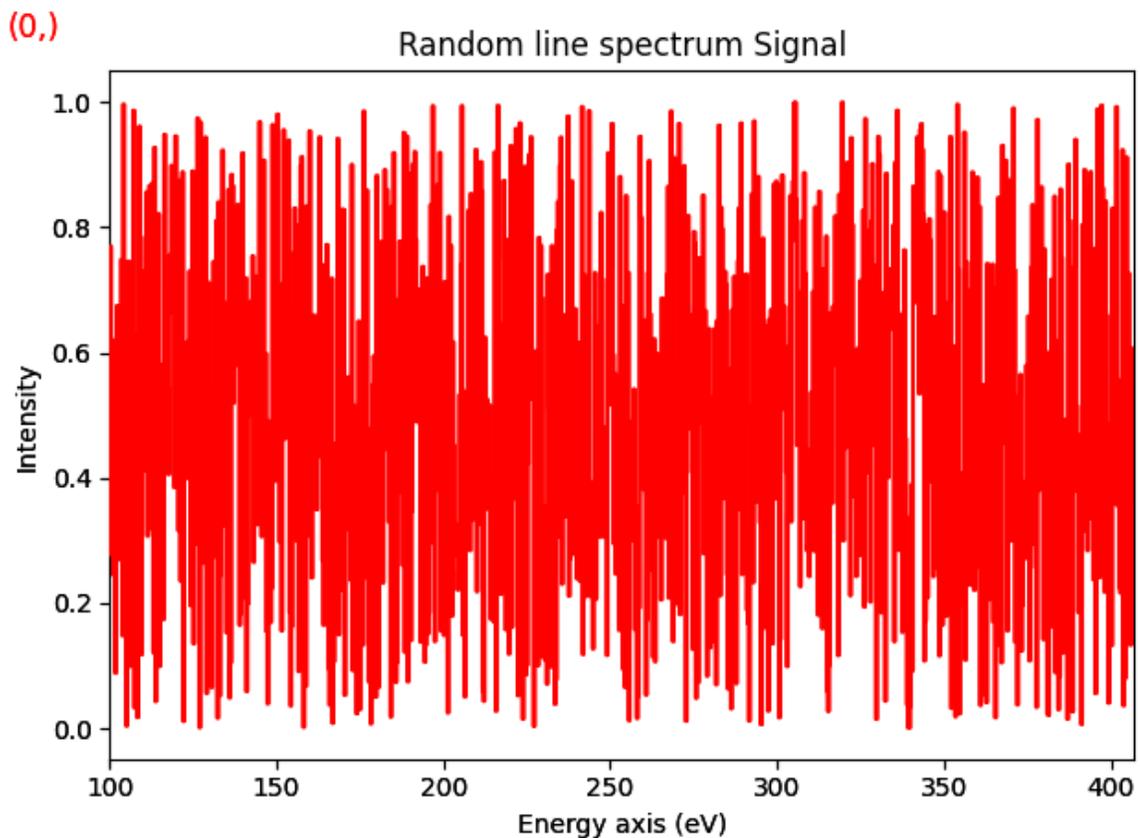
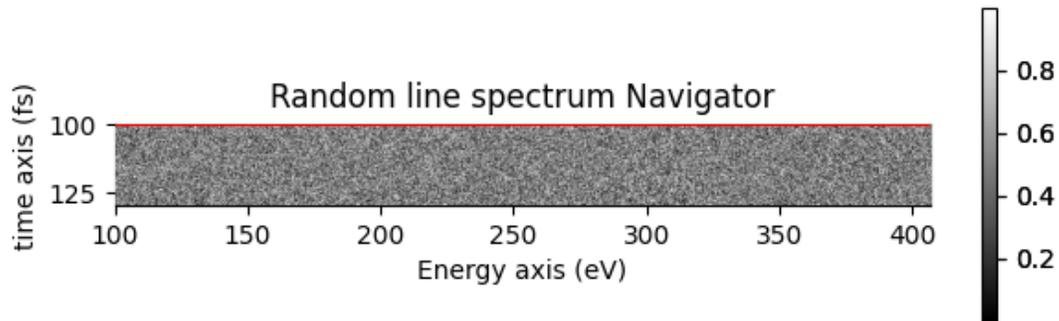
```
s.plot()
```



Total running time of the script: (0 minutes 0.321 seconds)

22.2.2 Creates a line spectrum

This example creates a line spectrum and plots it.



```
import numpy as np
import hyperspy.api as hs

# Create a line spectrum with random data
s = hs.signals.Signal1D(np.random.random((100, 1024)))

# Define the axis properties
s.axes_manager.signal_axes[0].name = 'Energy'
s.axes_manager.signal_axes[0].units = 'eV'
```

(continues on next page)

(continued from previous page)

```
s.axes_manager.signal_axes[0].scale = 0.3
s.axes_manager.signal_axes[0].offset = 100

s.axes_manager.navigation_axes[0].name = 'time'
s.axes_manager.navigation_axes[0].units = 'fs'
s.axes_manager.navigation_axes[0].scale = 0.3
s.axes_manager.navigation_axes[0].offset = 100

# Give a title
s.metadata.General.title = 'Random line spectrum'

# Plot it
s.plot()
```

Total running time of the script: (0 minutes 0.735 seconds)

22.2.3 Creates a signal1D from tabular data

This example creates a signal from tabular data, where the signal axis is given by an array of data values (the x column) and the tabular data are ordered in columns with 5 columns containing each 20 values and each column corresponding to a position in the navigation space (linescan).

```
import numpy as np
import hyperspy.api as hs
```

Create a set of tabular data:

```
x = np.linspace(0, 10, 20)
y = np.random.default_rng().random((20, 5))
```

Define the axes of the signal and then create the signal:

```
axes = [
    # length of the navigation axis
    dict(size=y.shape[1], scale=0.1, name="Position", units="nm"),
    # use values to define non-uniform axis for the signal axis
    dict(axis=x, name="Energy", units="eV"),
]

s = hs.signals.Signal1D(y.T, axes=axes)
```

Convert the non-uniform signal axis to a uniform axis, because non-uniform axes do not support all functionalities of HyperSpy. In this case, the error introduced during conversion to uniform *scale* is negligible.

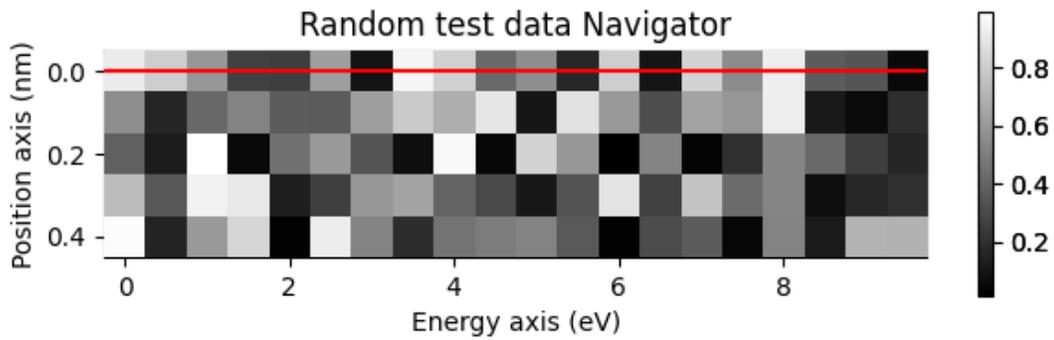
```
s.axes_manager.signal_axes[0].convert_to_uniform_axis()
```

Set title of the dataset and label for the data axis:

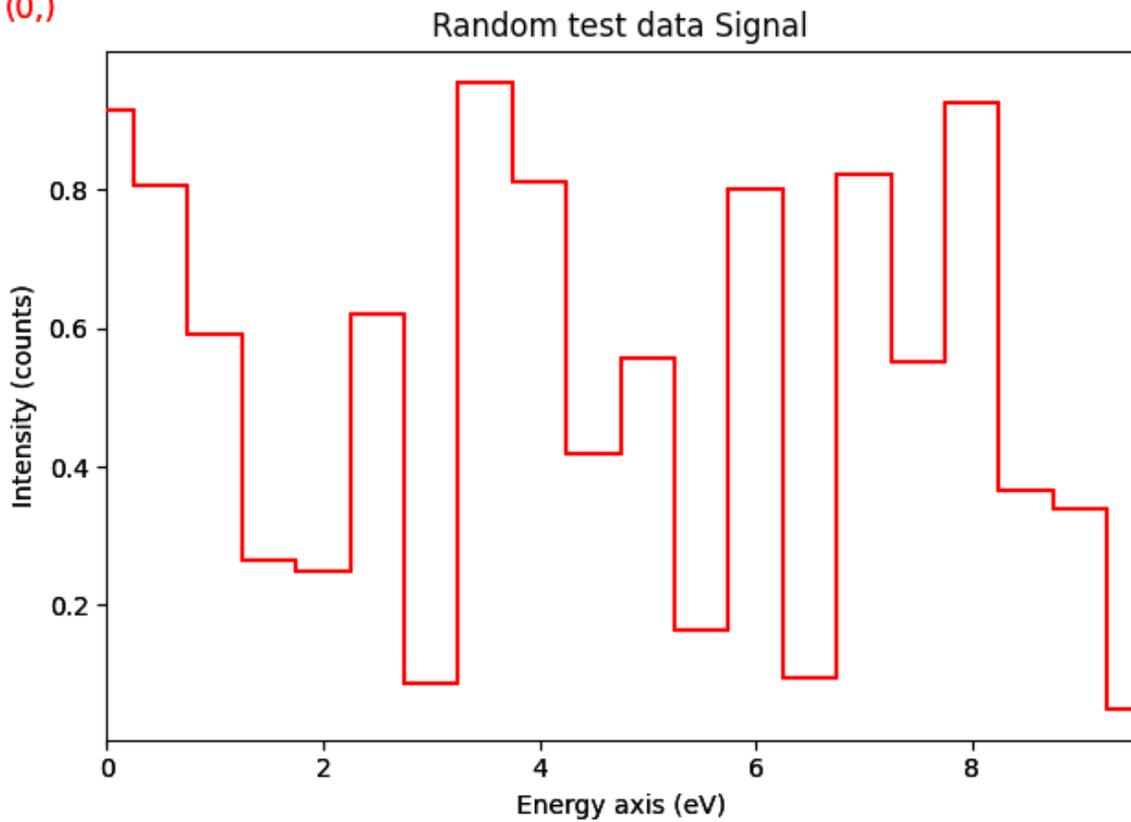
```
s.metadata.set_item("General.title", "Random test data")
s.metadata.set_item("Signal.quantity", "Intensity (counts)")
```

Plot the dataset:

```
s.plot()  
# Choose the second figure as gallery thumbnail:  
# sphinx_gallery_thumbnail_number = 2
```



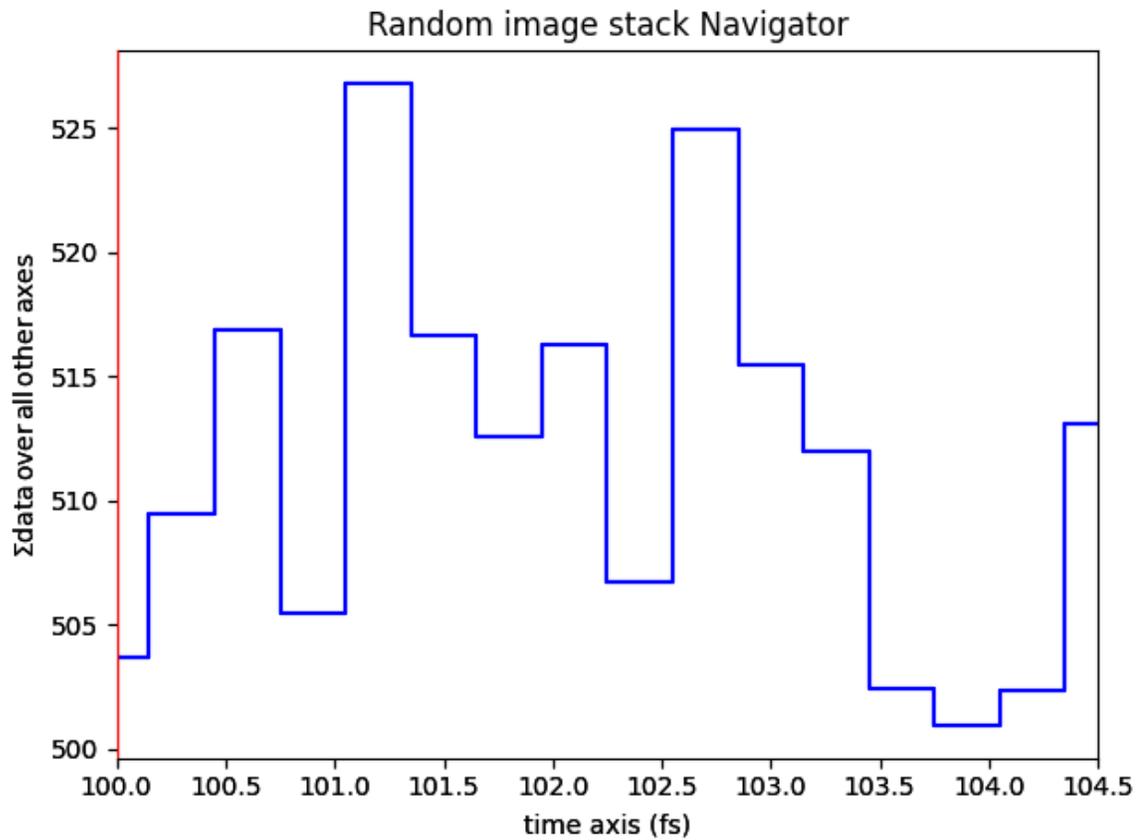
(0.)

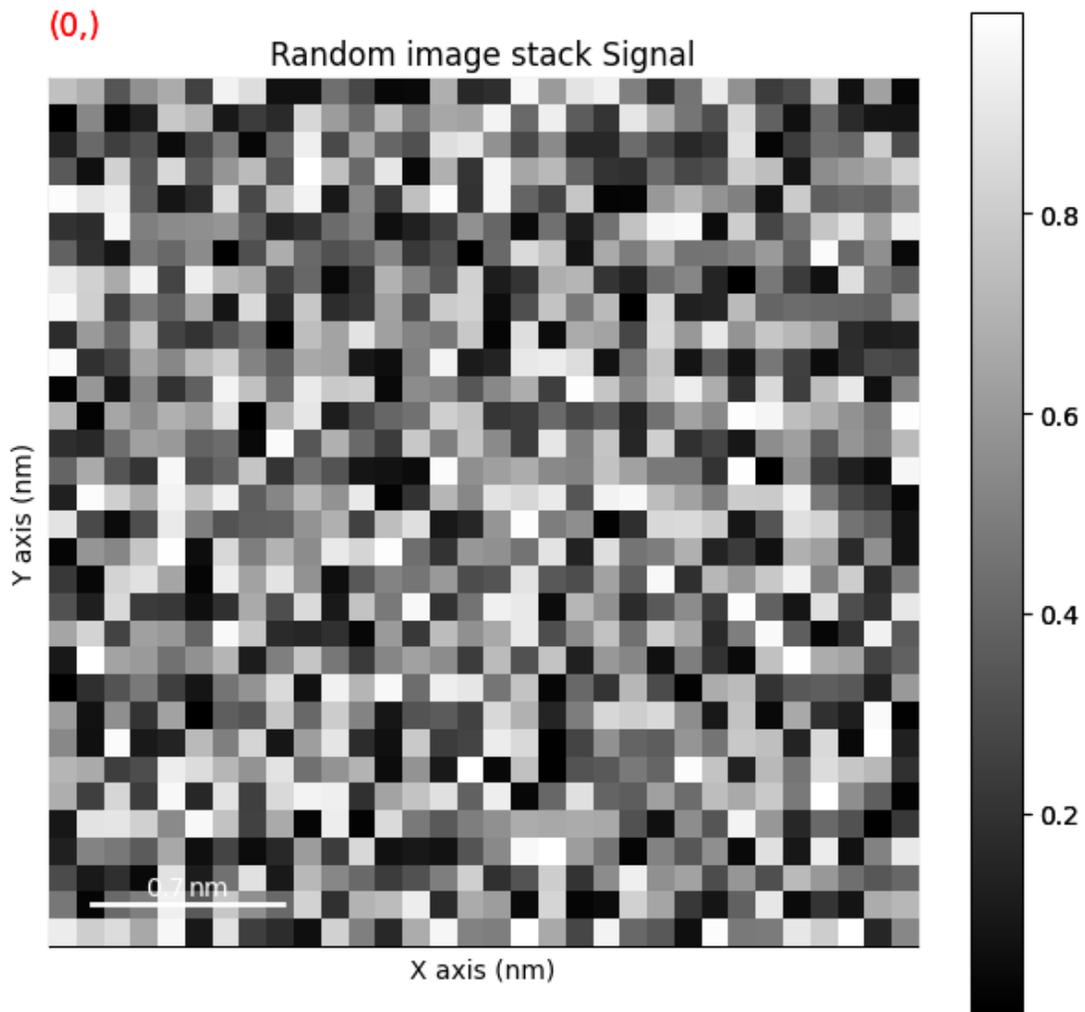


Total running time of the script: (0 minutes 0.587 seconds)

22.2.4 Creates a 3D image

This example creates an image stack and plots it.





```
import numpy as np
import hyperspy.api as hs

# Create an image stack with random data
im = hs.signals.Signal2D(np.random.random((16, 32, 32)))

# Define the axis properties
im.axes_manager.signal_axes[0].name = 'X'
im.axes_manager.signal_axes[0].units = 'nm'
im.axes_manager.signal_axes[0].scale = 0.1
im.axes_manager.signal_axes[0].offset = 0

im.axes_manager.signal_axes[1].name = 'Y'
im.axes_manager.signal_axes[1].units = 'nm'
im.axes_manager.signal_axes[1].scale = 0.1
im.axes_manager.signal_axes[1].offset = 0

im.axes_manager.navigation_axes[0].name = 'time'
im.axes_manager.navigation_axes[0].units = 'fs'
```

(continues on next page)

(continued from previous page)

```
im.axes_manager.navigation_axes[0].scale = 0.3
im.axes_manager.navigation_axes[0].offset = 100

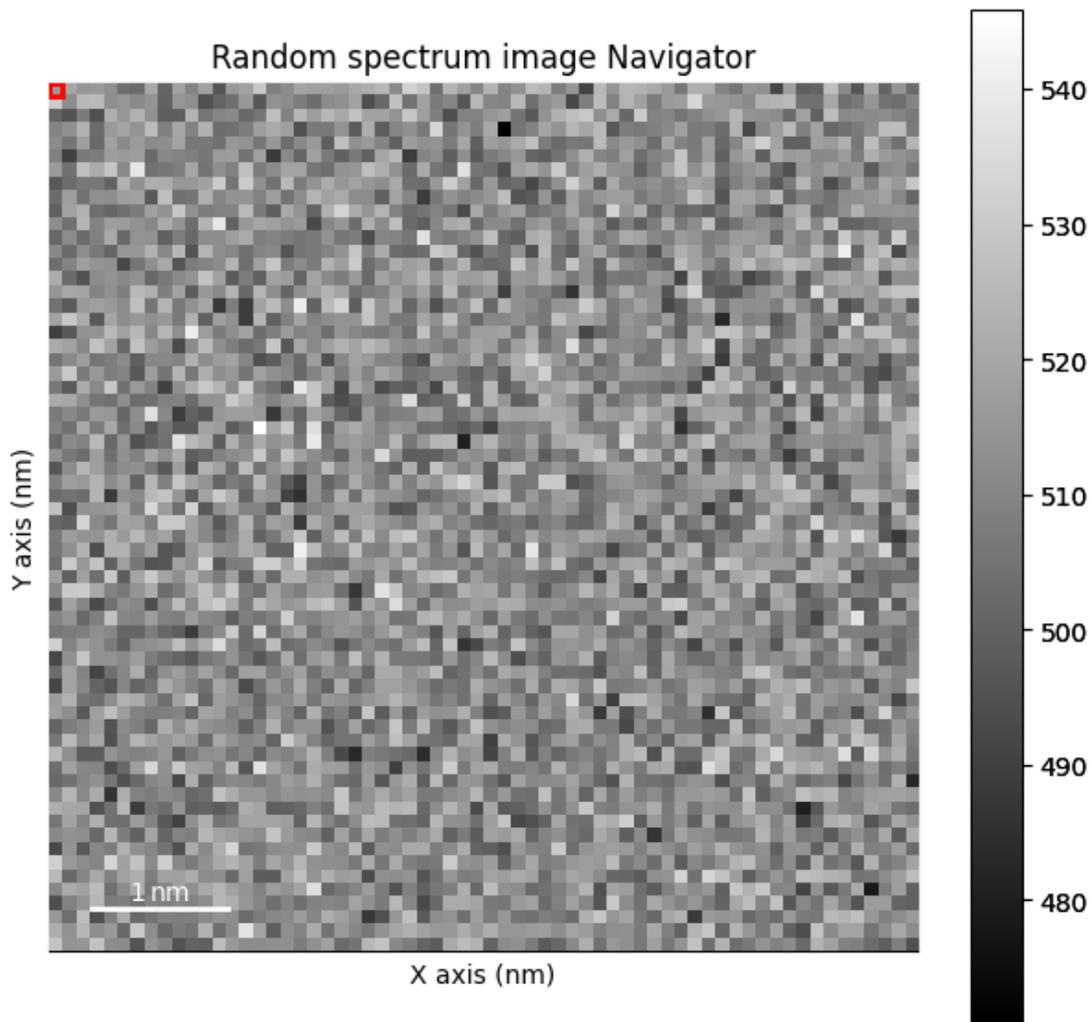
# Give a title
im.metadata.General.title = 'Random image stack'

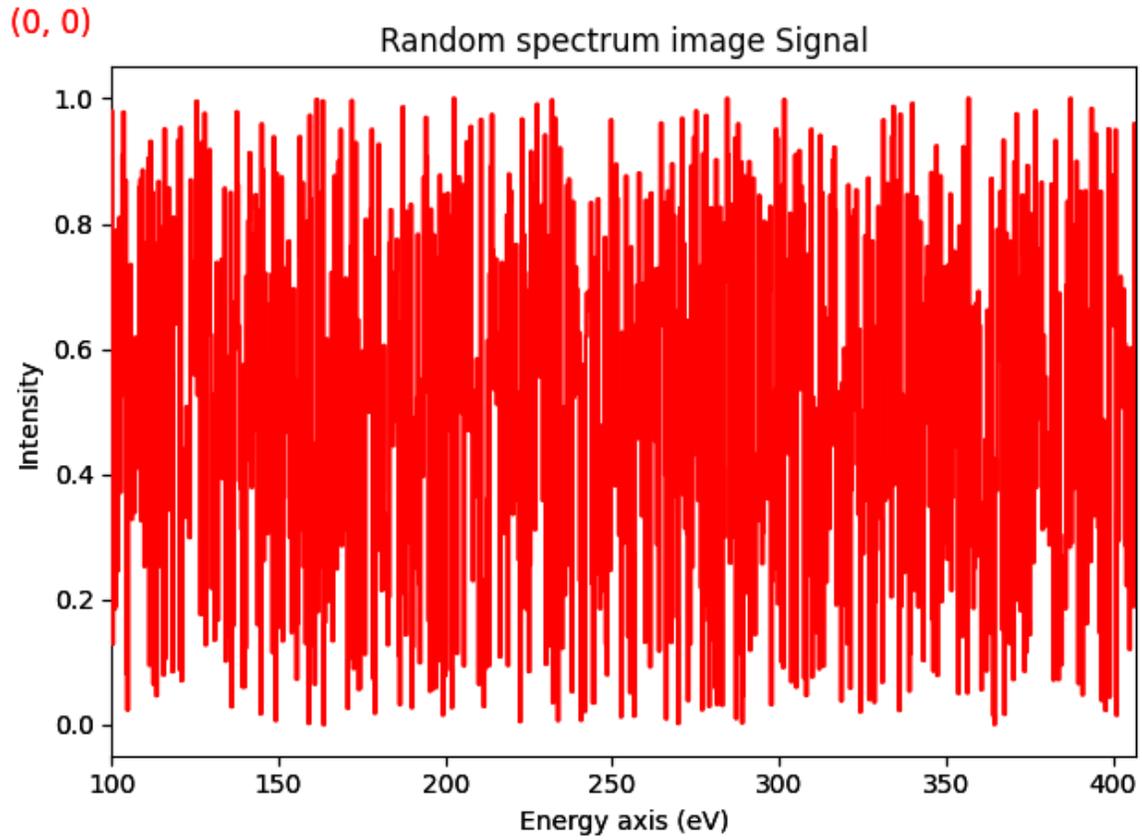
# Plot it
im.plot()
```

Total running time of the script: (0 minutes 0.660 seconds)

22.2.5 Creates a spectrum image

This example creates a spectrum image, i.e. navigation dimension 2 and signal dimension 1, and plots it.





```
import numpy as np
import hyperspy.api as hs
import matplotlib.pyplot as plt

# Create a spectrum image with random data
s = hs.signals.Signal1D(np.random.random((64, 64, 1024)))

# Define the axis properties
s.axes_manager.signal_axes[0].name = 'Energy'
s.axes_manager.signal_axes[0].units = 'eV'
s.axes_manager.signal_axes[0].scale = 0.3
s.axes_manager.signal_axes[0].offset = 100

s.axes_manager.navigation_axes[0].name = 'X'
s.axes_manager.navigation_axes[0].units = 'nm'
s.axes_manager.navigation_axes[0].scale = 0.1
s.axes_manager.navigation_axes[0].offset = 100

s.axes_manager.navigation_axes[1].name = 'Y'
s.axes_manager.navigation_axes[1].units = 'nm'
s.axes_manager.navigation_axes[1].scale = 0.1
s.axes_manager.navigation_axes[1].offset = 100

# Give a title
```

(continues on next page)

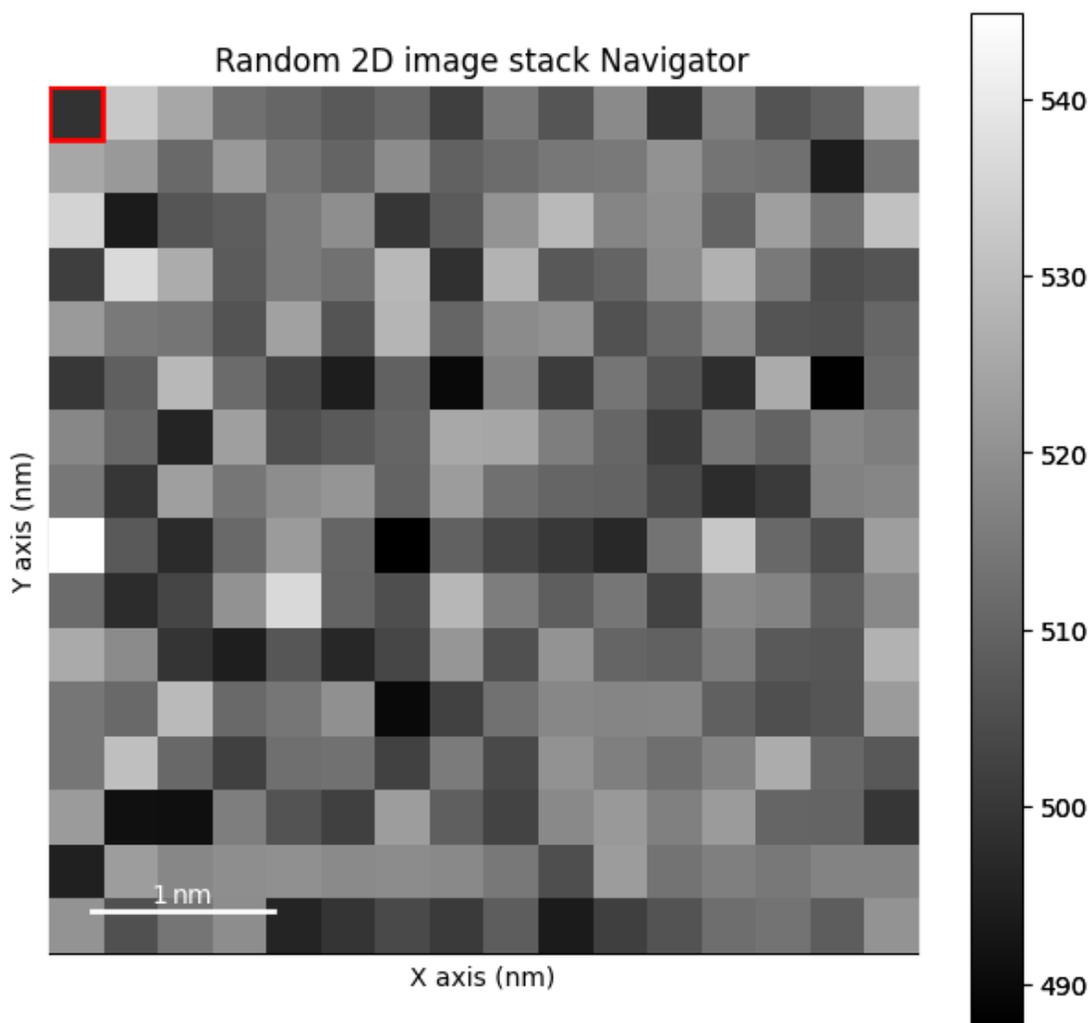
(continued from previous page)

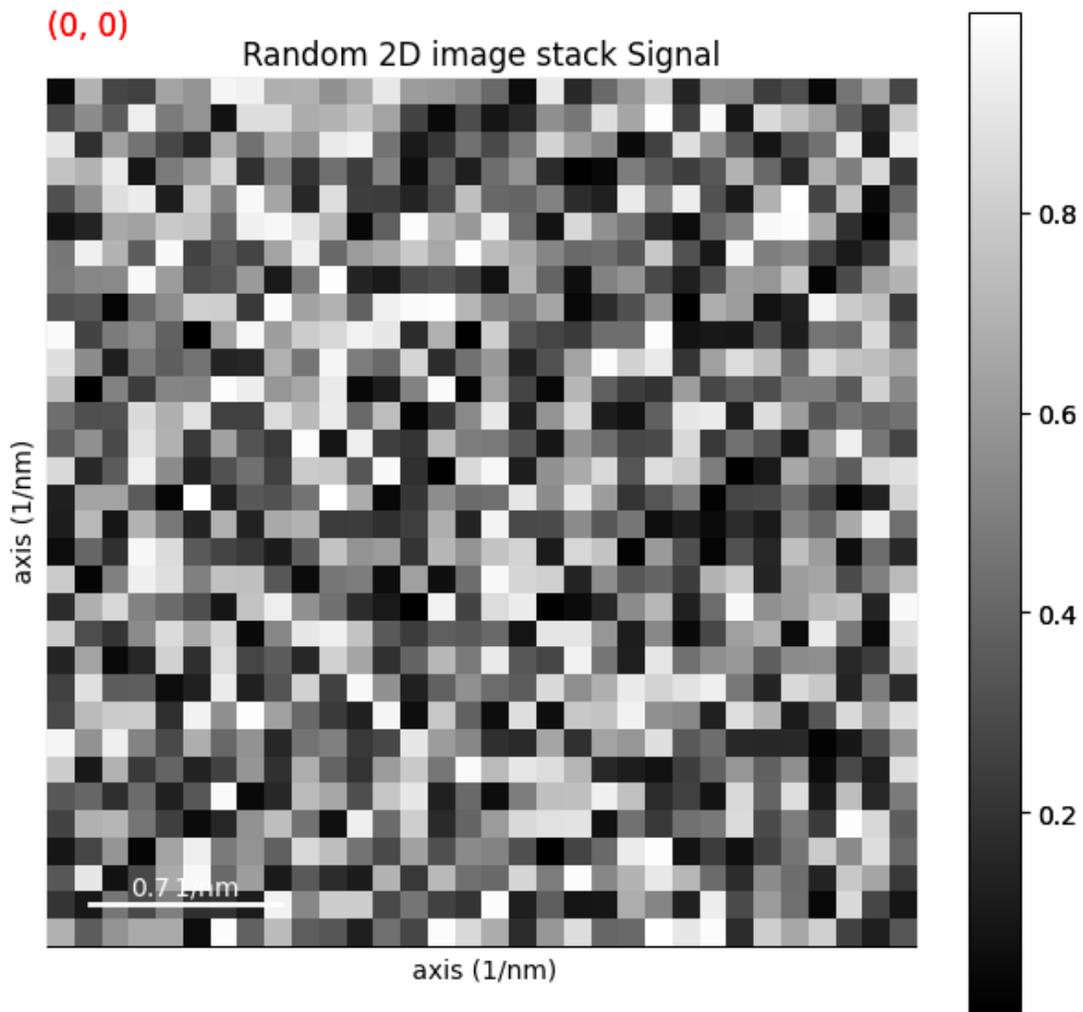
```
s.metadata.General.title = 'Random spectrum image'  
  
# Plot it  
s.plot()  
  
plt.show() # No necessary when running in the HyperSpy's IPython profile
```

Total running time of the script: (0 minutes 0.763 seconds)

22.2.6 Creates a 4D image

This example creates a 4D dataset, i.e. 2 navigation dimension and 2 signal dimension and plots it.





```
import numpy as np
import hyperspy.api as hs

# Create a 2D image stack with random data
im = hs.signals.Signal2D(np.random.random((16, 16, 32, 32)))

# Define the axis properties
im.axes_manager.signal_axes[0].name = ''
im.axes_manager.signal_axes[0].units = '1/nm'
im.axes_manager.signal_axes[0].scale = 0.1
im.axes_manager.signal_axes[0].offset = 0

im.axes_manager.signal_axes[1].name = ''
im.axes_manager.signal_axes[1].units = '1/nm'
im.axes_manager.signal_axes[1].scale = 0.1
im.axes_manager.signal_axes[1].offset = 0

im.axes_manager.navigation_axes[0].name = 'X'
im.axes_manager.navigation_axes[0].units = 'nm'
```

(continues on next page)

(continued from previous page)

```
im.axes_manager.navigation_axes[0].scale = 0.3
im.axes_manager.navigation_axes[0].offset = 100

im.axes_manager.navigation_axes[1].name = 'Y'
im.axes_manager.navigation_axes[1].units = 'nm'
im.axes_manager.navigation_axes[1].scale = 0.3
im.axes_manager.navigation_axes[1].offset = 100

# Give a title
im.metadata.General.title = 'Random 2D image stack'

im.plot()
```

Total running time of the script: (0 minutes 0.638 seconds)

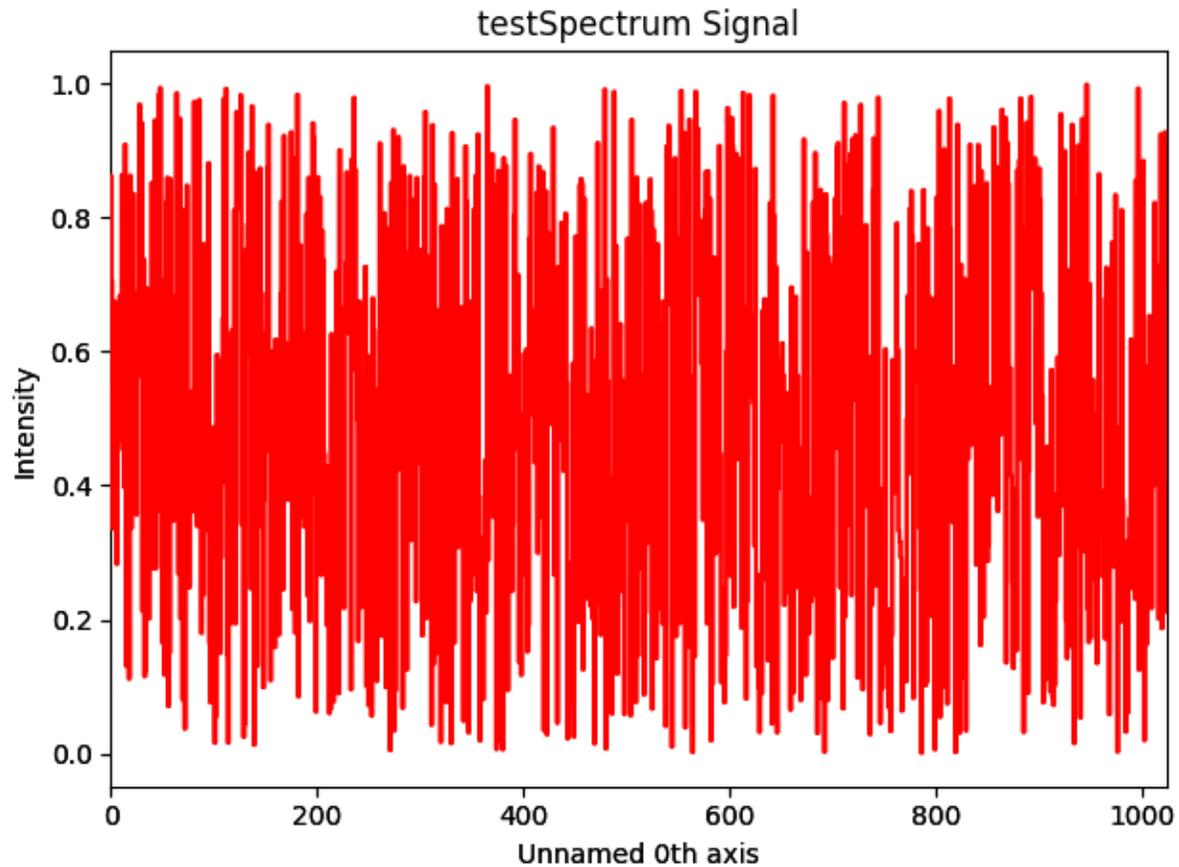
22.3 Loading, saving and exporting

Below is a gallery of examples on loading, saving and exporting data.

22.3.1 Export single spectrum

Creates a single spectrum image, saves it and plots it:

1. Create a single spectrum using *SignalID* signal.
2. Save signal as a msa file
3. Plot the signal using the *plot* method
4. Save the figure as a png file



```
# Set the matplotlib backend of your choice, for example
# %matplotlib qt
import hyperspy.api as hs
import numpy as np

s = hs.signals.Signal1D(np.random.rand(1024))

# Export as msa file, very similar to a csv file but containing standardised
# metadata
s.save('testSpectrum.msa', overwrite=True)

# Plot it
s.plot()
```

Total running time of the script: (0 minutes 0.354 seconds)

22.4 Model fitting

Below is a gallery of examples on model fitting.

22.4.1 Plot Residual

Fit an affine function and plot the residual.

```
import numpy as np
import hyperspy.api as hs
```

Create a signal:

```
data = np.arange(1000, dtype=np.int64).reshape((10, 100))
s = hs.signals.Signal1D(data)
```

Add noise:

```
s.add_poissonian_noise(random_state=0)
```

Create model:

```
m = s.create_model()
line = hs.model.components1D.Expression("a * x + b", name="Affine")
m.append(line)
```

Fit for all navigation positions:

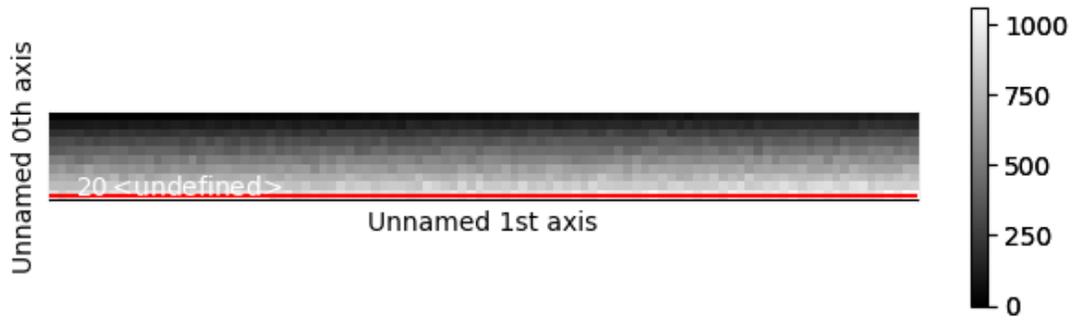
```
m.multifit()
```

```
Exception ignored in: <function tqdm.__del__ at 0x7f3e1ecc37e0>
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/hyperspy/envs/latest/lib/
python3.11/site-packages/tqdm/std.py", line 1148, in __del__
    self.close()
  File "/home/docs/checkouts/readthedocs.org/user_builds/hyperspy/envs/latest/lib/
python3.11/site-packages/tqdm/notebook.py", line 279, in close
    self.disp(bar_style='danger', check_delay=False)
    ^^^^^^^^^^^
AttributeError: 'tqdm_notebook' object has no attribute 'disp'

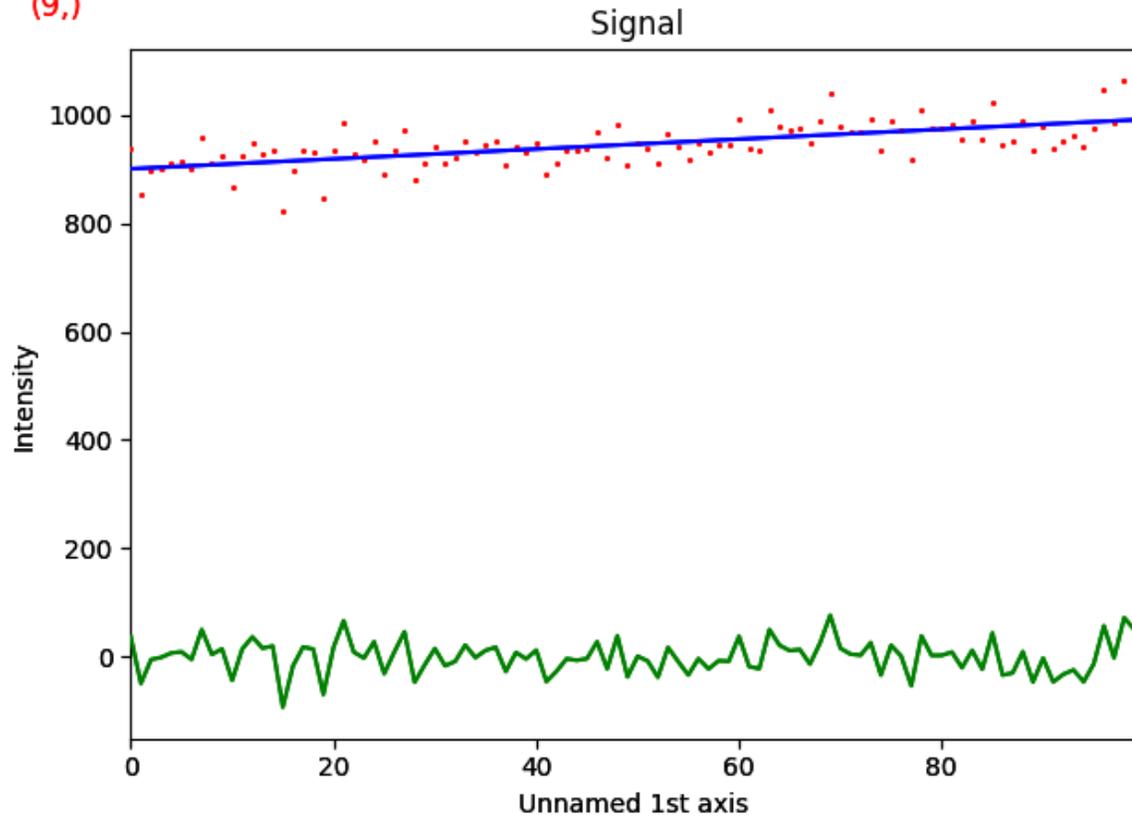
 0%|          | 0/10 [00:00<?, ?it/s]
100%|| 10/10 [00:00<00:00, 1029.23it/s]
```

Plot the fitted model with residual:

```
m.plot(plot_residual=True)
# Choose the second figure as gallery thumbnail:
# sphinx_gallery_thumbnail_number = 2
```



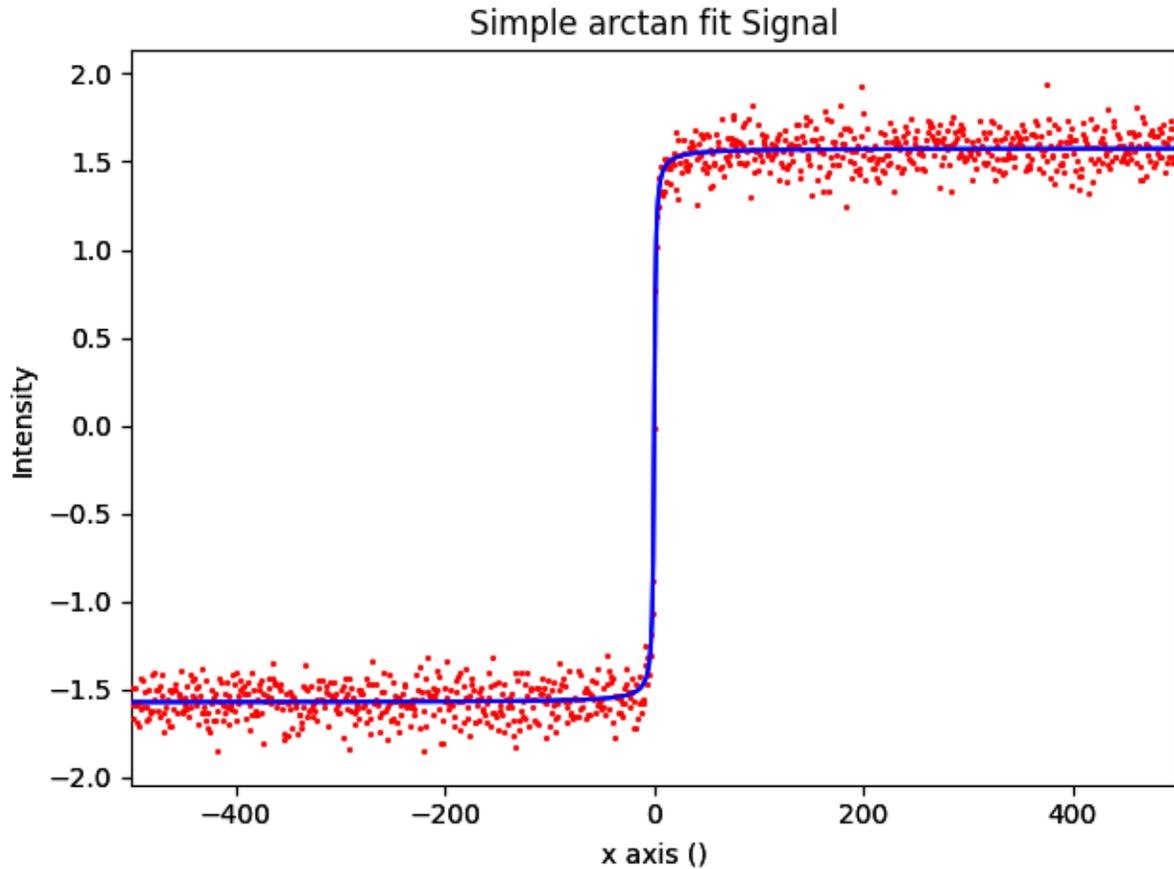
(9,)



Total running time of the script: (0 minutes 0.711 seconds)

22.4.2 Simple arctan fit

Fit an arctan function.



ModelID: Simple arctan fit

CurrentComponentValues: Arctan

Active: True

Parameter Name	Free	Value	Std	Min	Max	Linear
A	True	1.00375570	0.00213427	None	None	True
k	True	0.91254693	0.07331322	None	None	False
x0	True	0.06132433	0.08301128	None	None	False

```
import numpy as np
import hyperspy.api as hs

# Generate the data and make the spectrum
data = np.arctan(np.arange(-500, 500))
s = hs.signals.Signal1D(data)
```

(continues on next page)

(continued from previous page)

```

s.axes_manager[0].offset = -500
s.axes_manager[0].units = ""
s.axes_manager[0].name = "x"
s.metadata.General.title = "Simple arctan fit"
s.set_signal_origin("simulation")

s.add_gaussian_noise(0.1)

# Make the arctan component for use in the model
arctan_component = hs.model.components1D.Arctan()

# Create the model and add the arctan component
m = s.create_model()
m.append(arctan_component)

# Fit the arctan component to the spectrum
m.fit()

# Print the result of the fit
m.print_current_values()

# Plot the spectrum and the model fitting
m.plot()

```

Total running time of the script: (0 minutes 0.554 seconds)

22.5 Region of Interest

Below is a gallery of examples on using regions of interest with HyperSpy signals.

22.5.1 SpanROI on signal axis

Use a SpanROI interactively on a Signal1D.

```
import hyperspy.api as hs
```

Create a signal:

```
s = hs.data.two_gaussians()
```

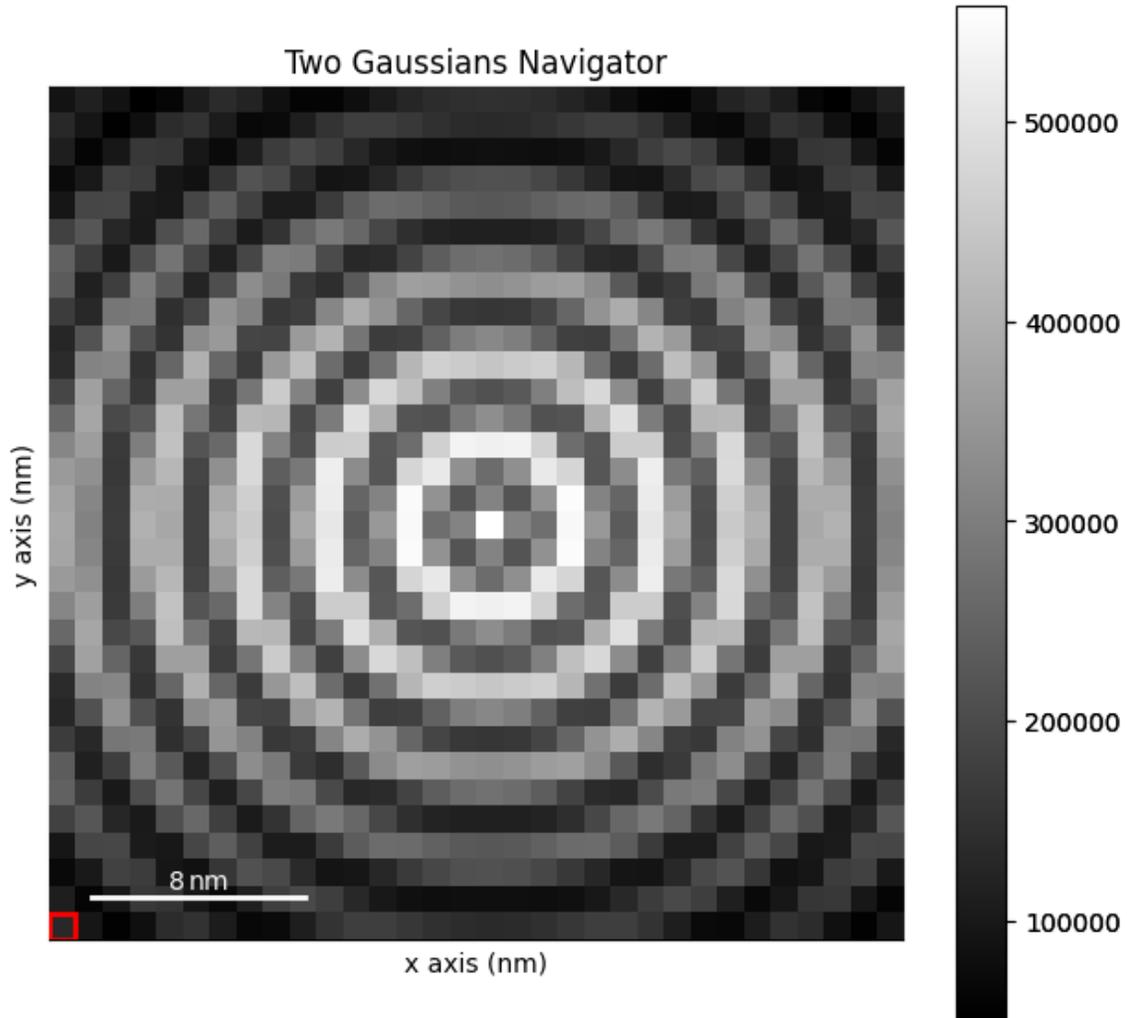
Create the roi, here a *SpanROI* for one dimensional ROI:

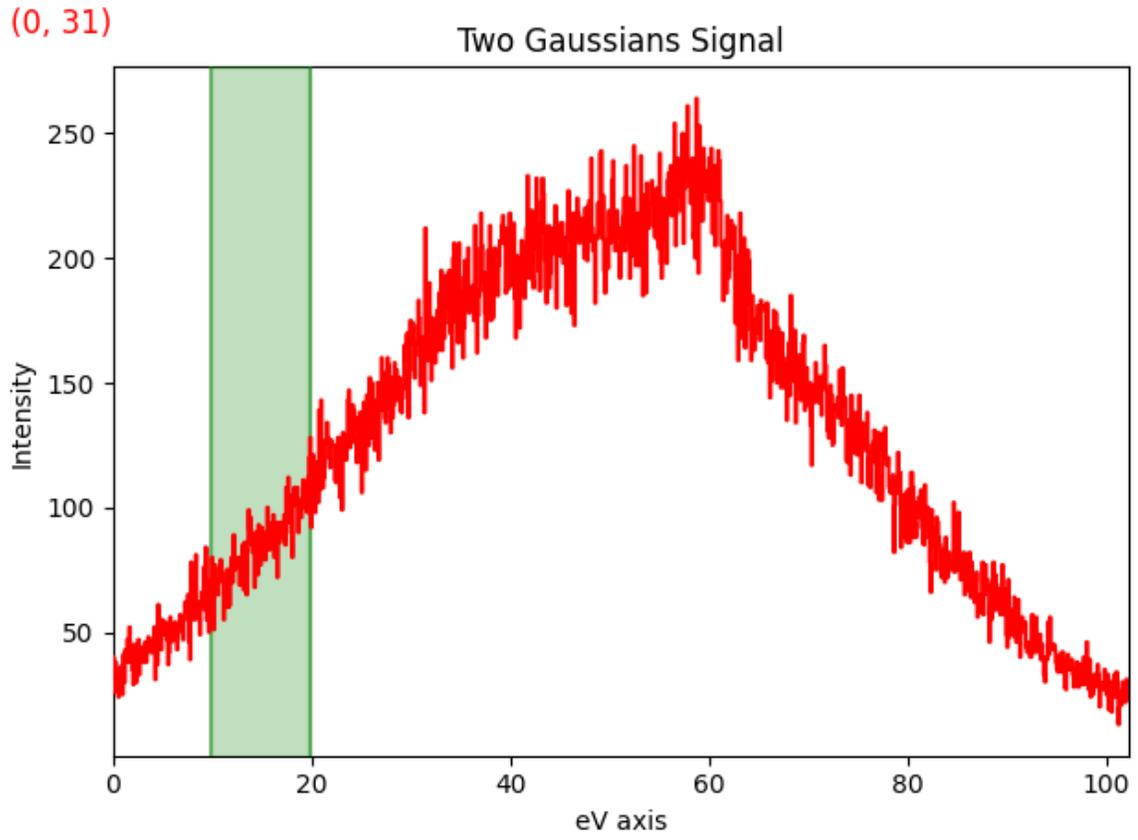
```
roi = hs.roi.SpanROI(left=10, right=20)
```

Slice signal with roi with the ROI. By using the *interactive* function, the output signal `s_roi` will update automatically. The ROI will be added automatically on the signal figure.

Specify the axes to add the ROI on either the navigation or signal dimension:

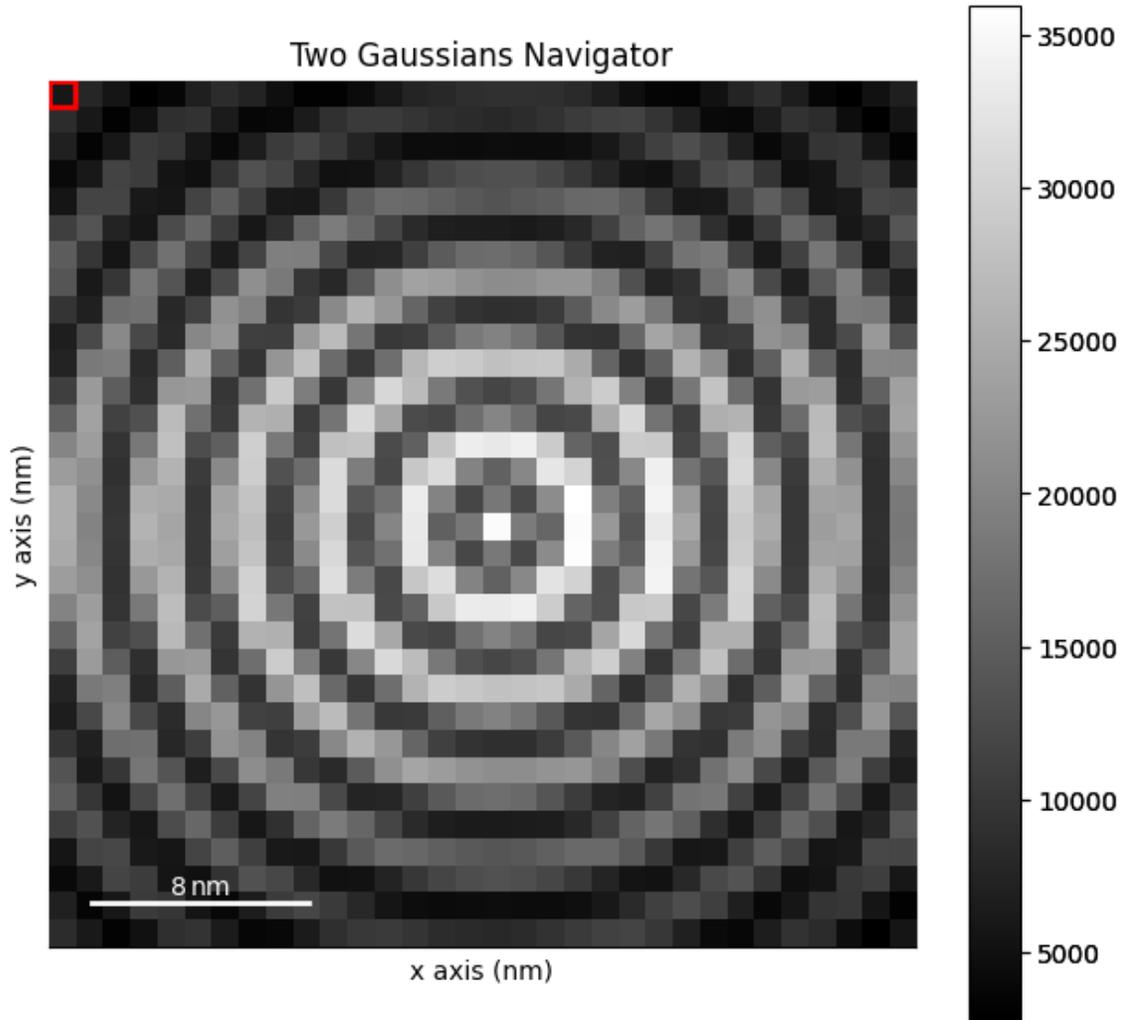
```
s.plot()
sliced_signal = roi.interactive(s, axes=s.axes_manager.signal_axes)
# Choose the second figure as gallery thumbnail:
# sphinx_gallery_thumbnail_number = 2
```

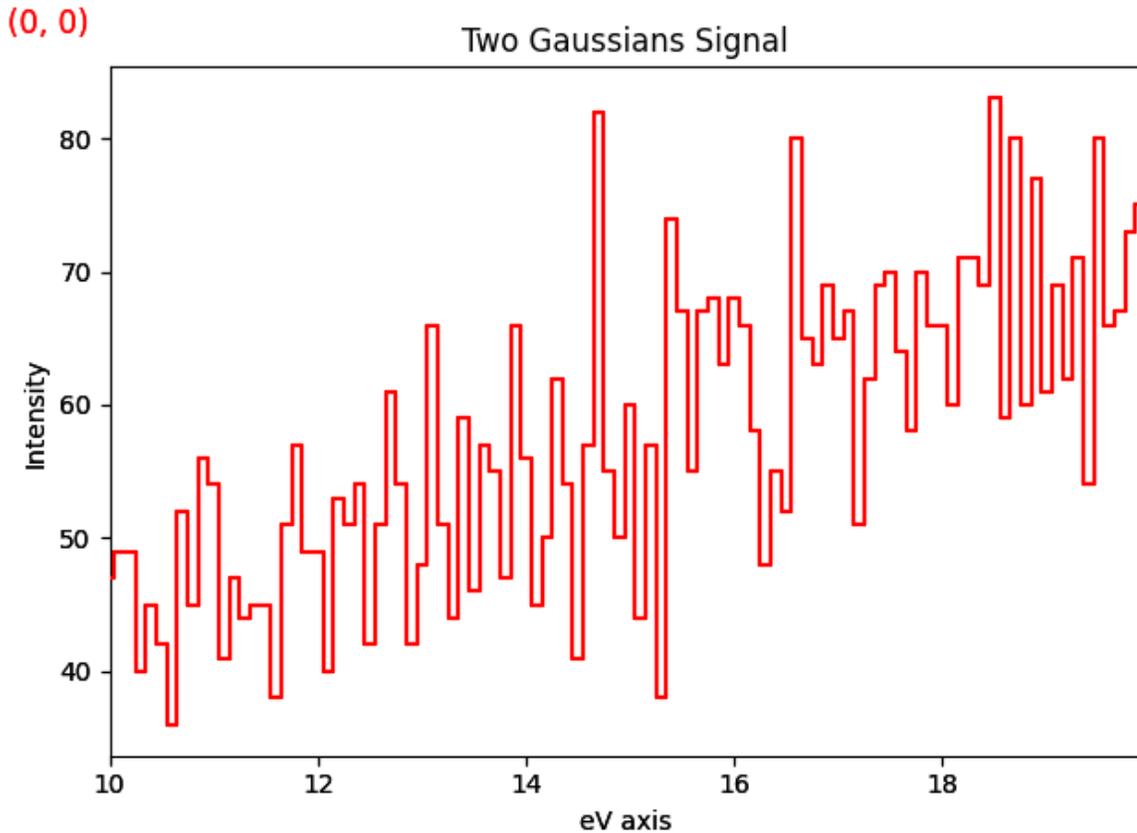




Plot the signal sliced by the ROI and use `autoscale='xv'` to update the limits of the plot automatically:

```
sliced_signal.plot(autoscale='xv')
```





Total running time of the script: (0 minutes 2.104 seconds)

22.5.2 Navigator ROI

Use a `RectangularROI` to take the sum of an area of the navigation space.

```
import hyperspy.api as hs
```

Create a signal:

```
s = hs.data.two_gaussians()
```

Create the roi, here a `RectangularROI` for the two dimension navigation space:

```
roi = hs.roi.RectangularROI()
```

Slice signal with roi with the ROI. By using the `interactive` function, the output signal `s_roi` will update automatically. The ROI will be added automatically on the signal figure.

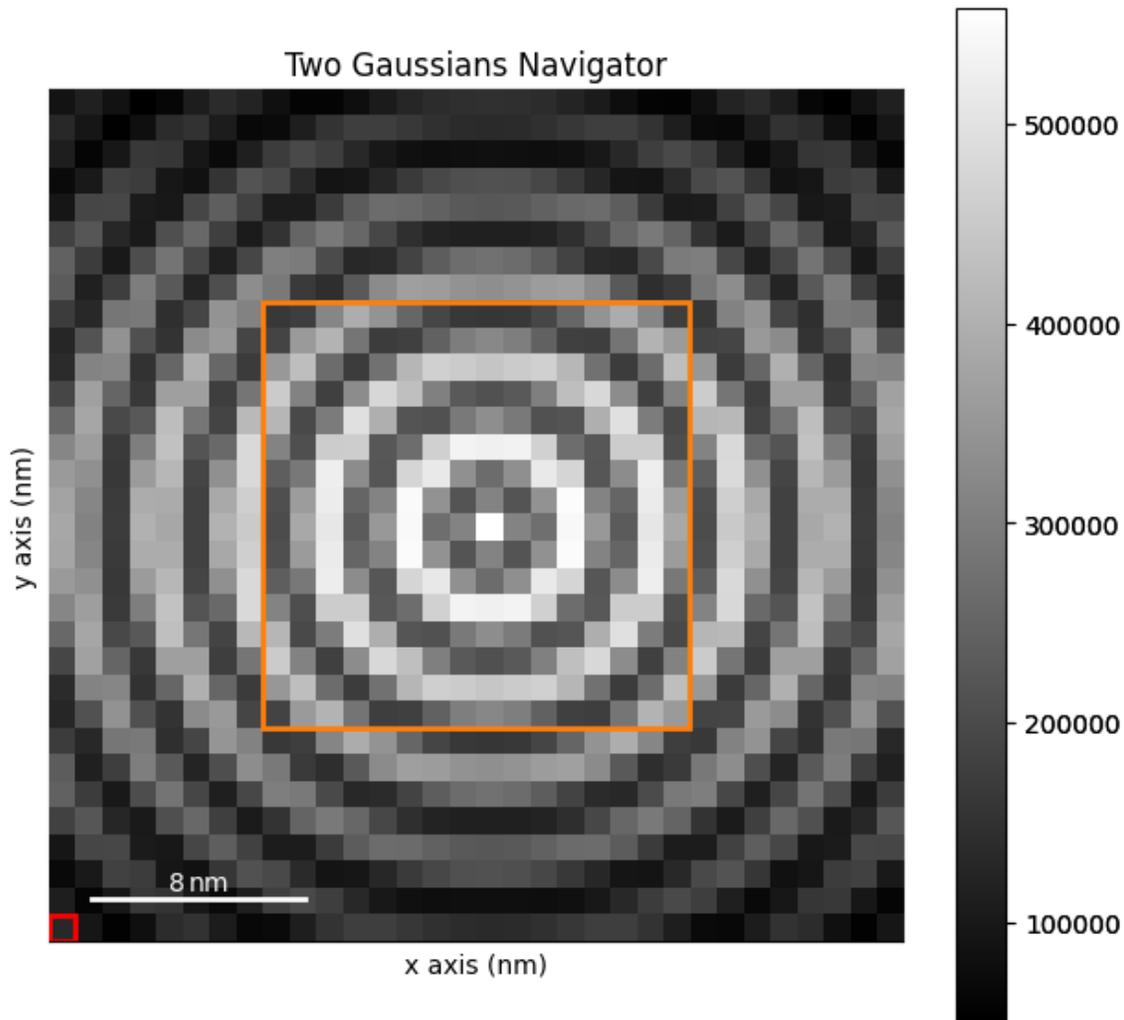
By default, the ROI will be added to the navigation or signal. We specify `recompute_out_event=None` to avoid redundant computation when changing the ROI

```
s.plot()
s_roi = roi.interactive(s, recompute_out_event=None, color='C1')
```

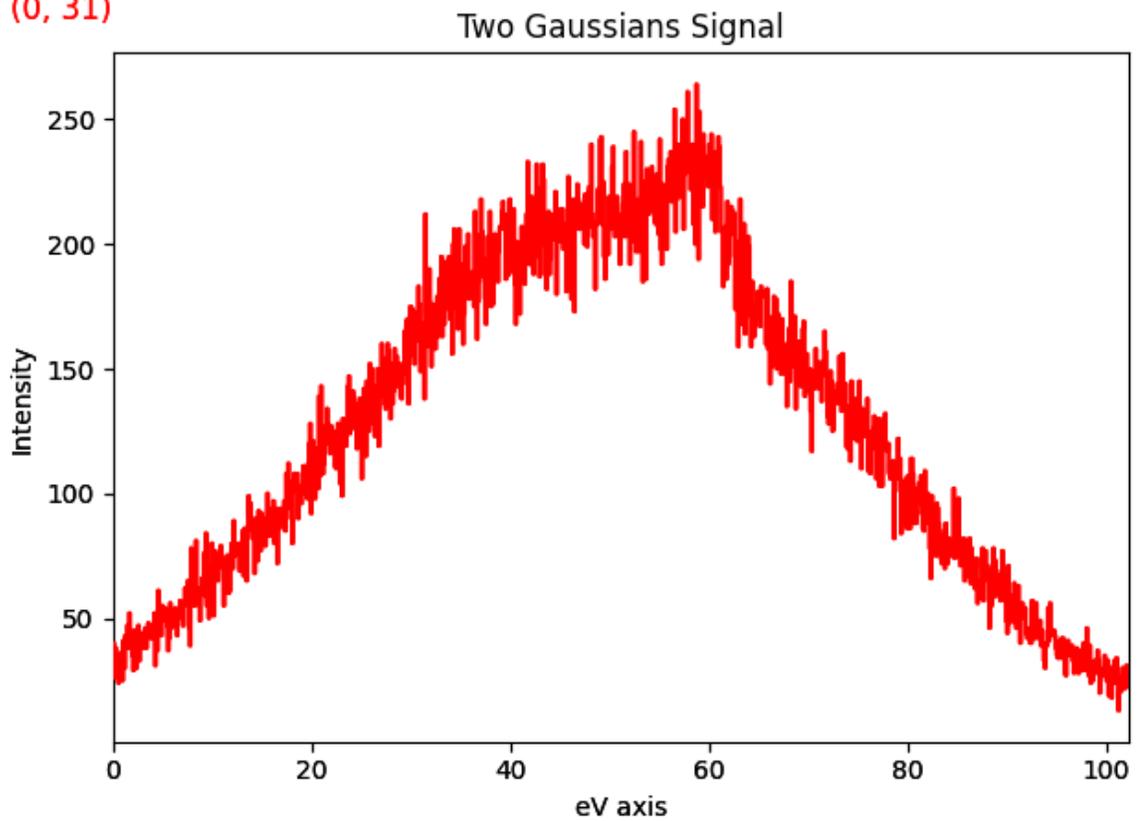
(continues on next page)

(continued from previous page)

```
# We use :py:class:`~.interactive` function to compute the sum over the ROI interactively:  
roi_sum = hs.interactive(s_roi.sum, recompute_out_event=None)  
  
# Choose the second figure as gallery thumbnail:  
# sphinx_gallery_thumbnail_number = 1
```

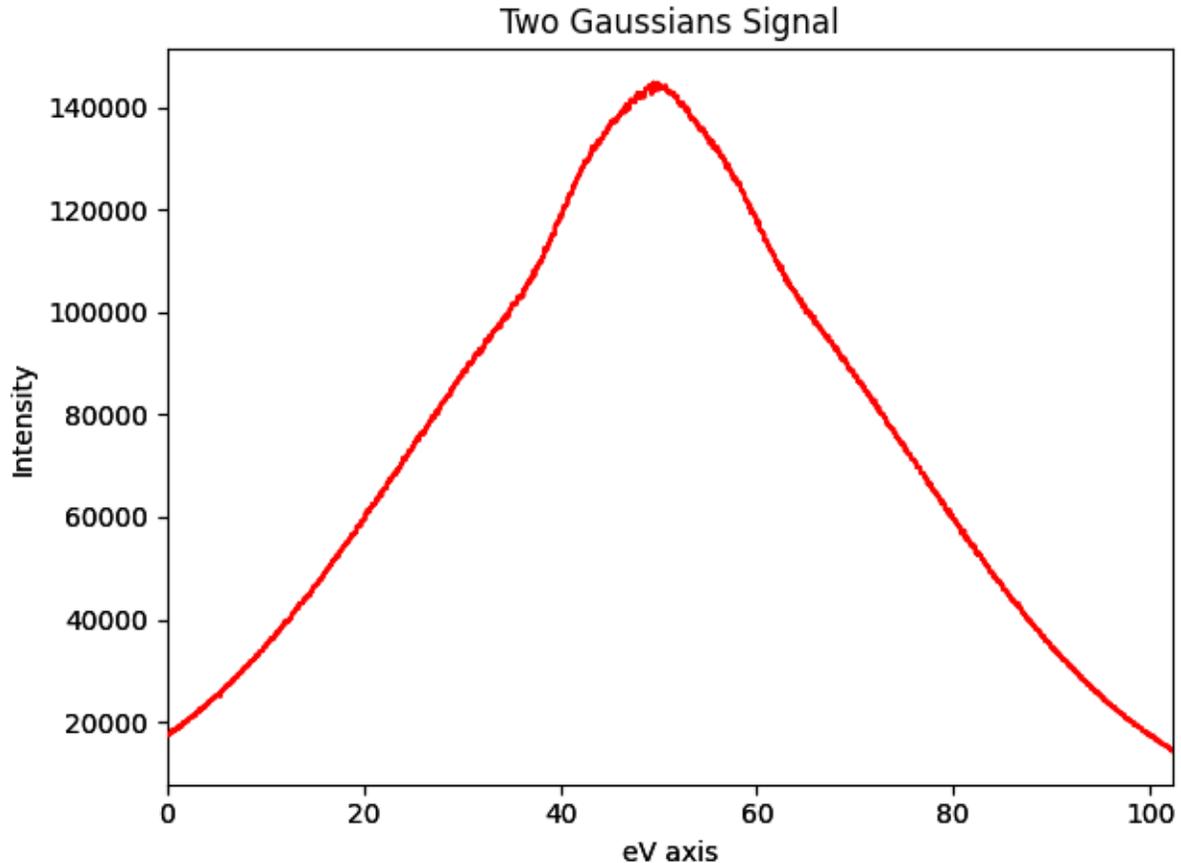


(0, 31)



Plot the signal sliced by the ROI:

```
roi_sum.plot()
```



Total running time of the script: (0 minutes 1.568 seconds)

22.5.3 Extract line profile from image interactively

Interactively extract a line profile (with a certain width) from an image using `Line2DROI`. Use `plot_spectra()` to plot several line profiles on the same figure. Save a profile data as msa file.

Fig. 1: Extracting line profiles and interactive plotting.

Initialize image data as HyperSpy signal:

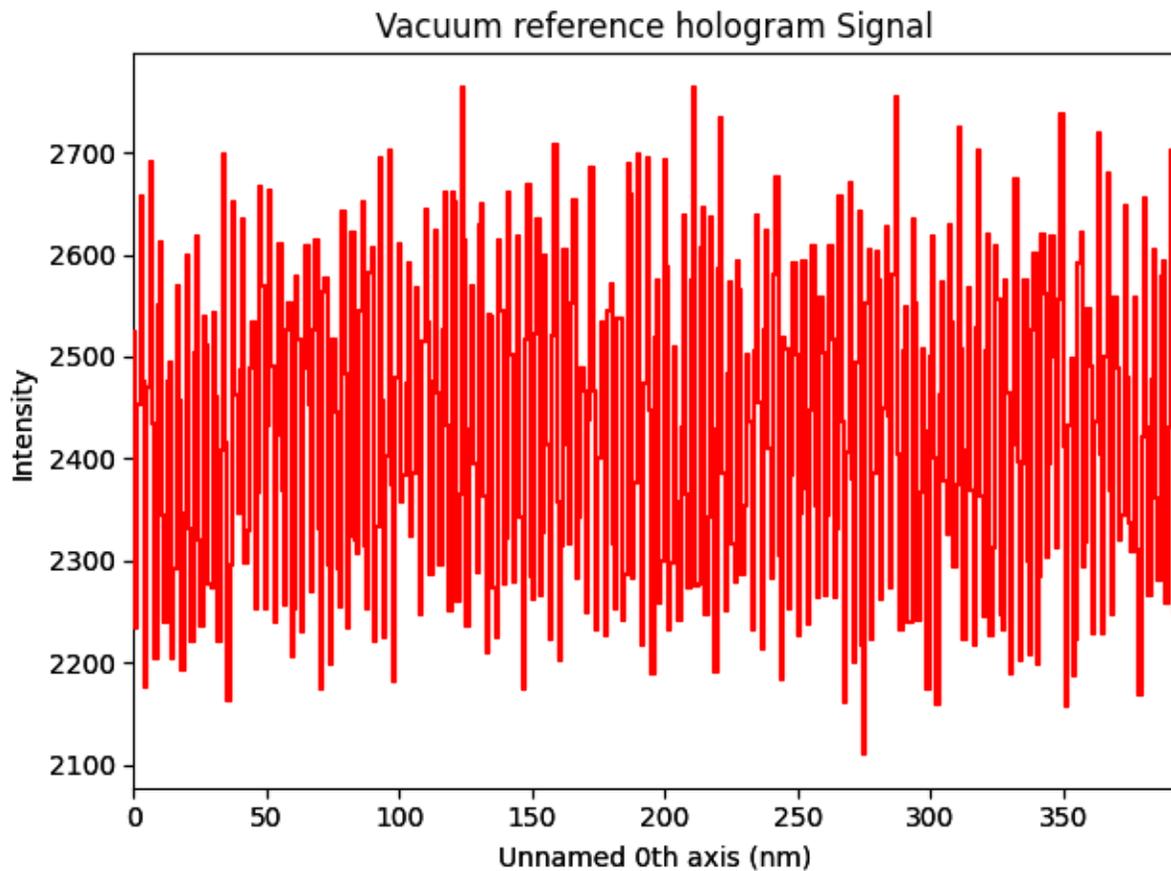
```
import hyperspy.api as hs
import holospy as holo
im0 = holo.data.Fe_needle_reference_hologram()
im1 = holo.data.Fe_needle_hologram()
```

Initialize Line-ROI from position (400,250) to position (220,600) of width 5 in calibrated axes units (in the current example equal to the image pixels):

```
line_roi = hs.roi.Line2DROI(400, 250, 220, 600, 5)
```

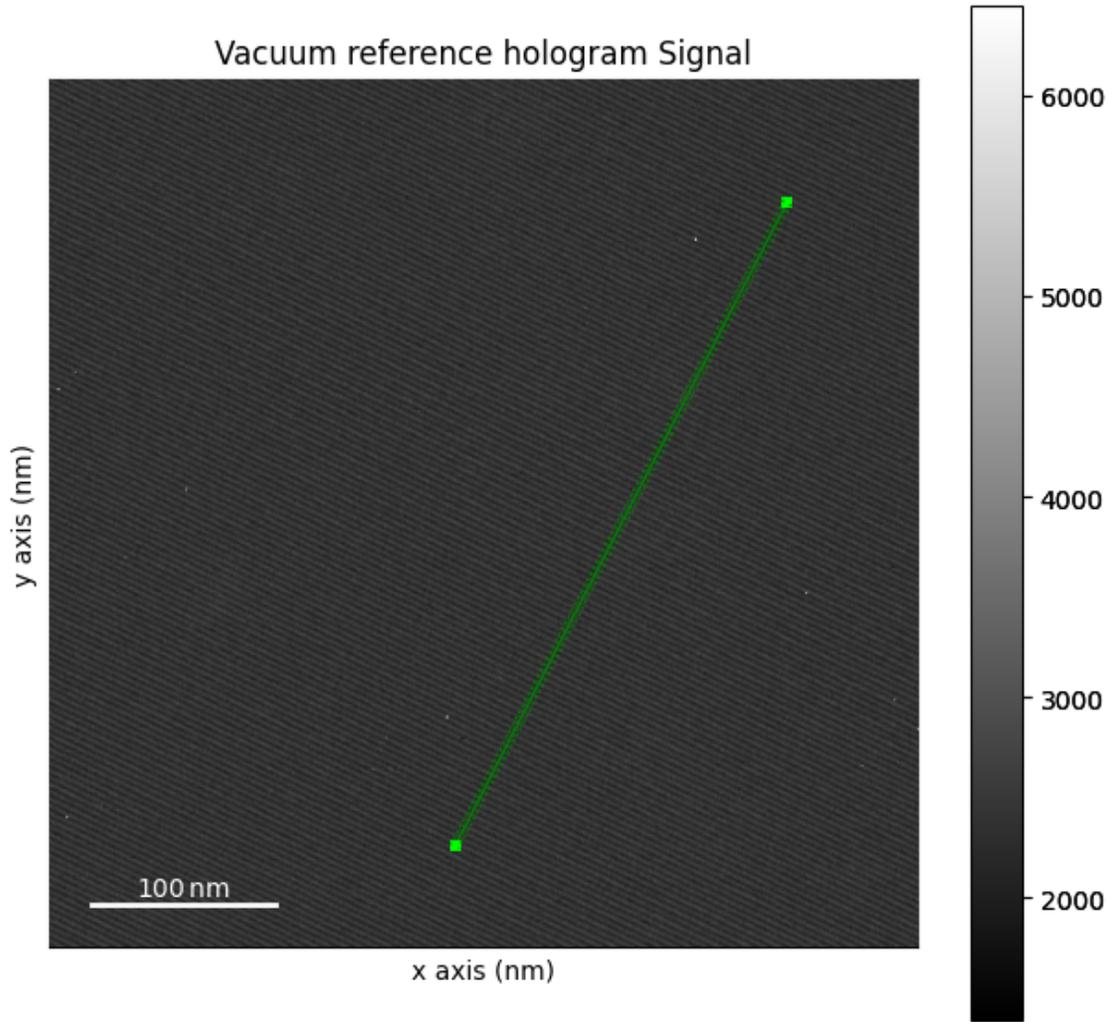
Extract data along the ROI as new signal by “slicing” the signal and plot the profile:

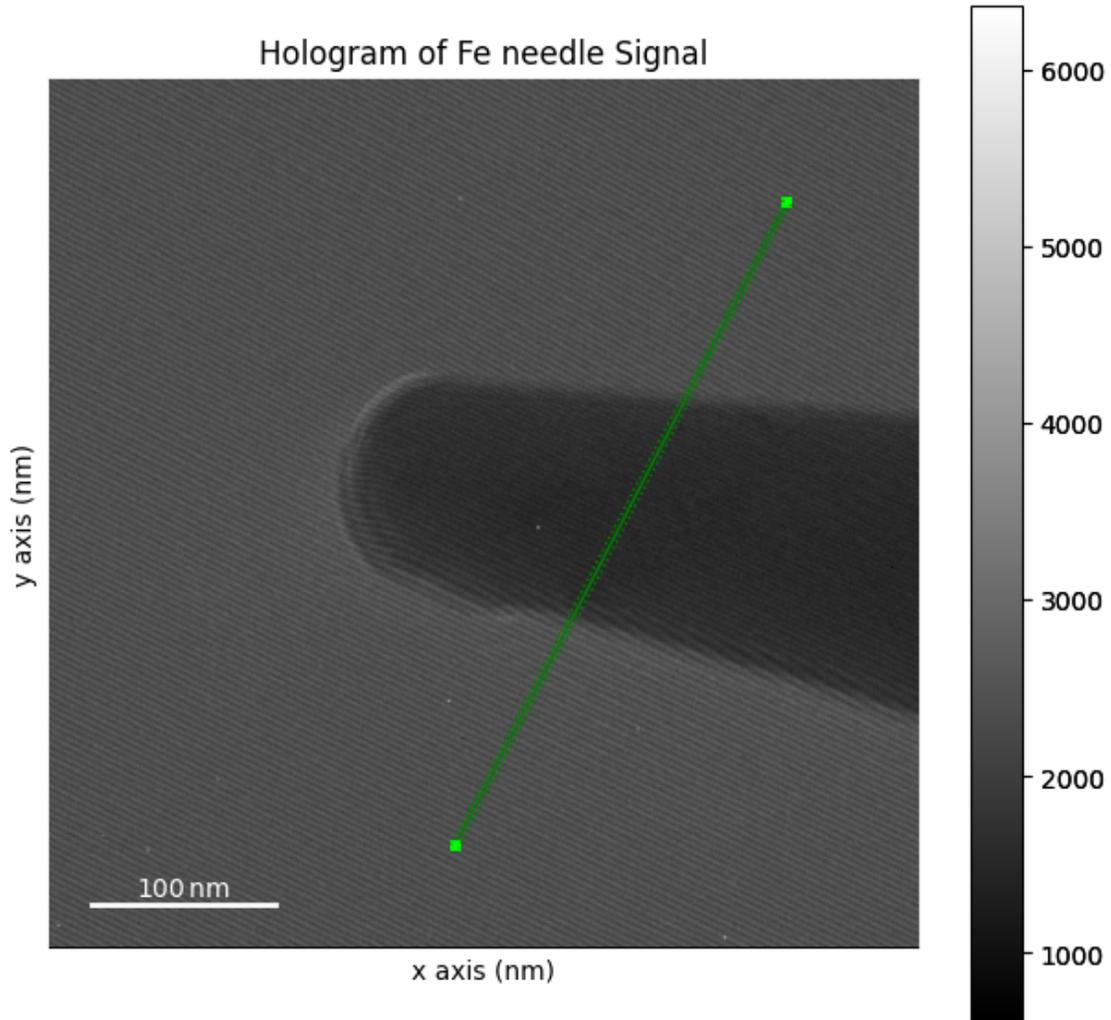
```
profile = line_roi(im0)
profile.plot()
```



Slicing of the signal is not interactive. If you want to modify the line along which the profile is extracted, you can plot the image and display the ROI interactively (creates a new signal object). You can even display the same ROI on a second image to make sure that a profile is well placed on both images:

```
im0.plot()
profile1 = line_roi.interactive(im0, color='green')
im1.plot()
profile2 = line_roi.interactive(im1, color='green')
```





You can then drag and drop the ends of the ROI to adjust the placement.

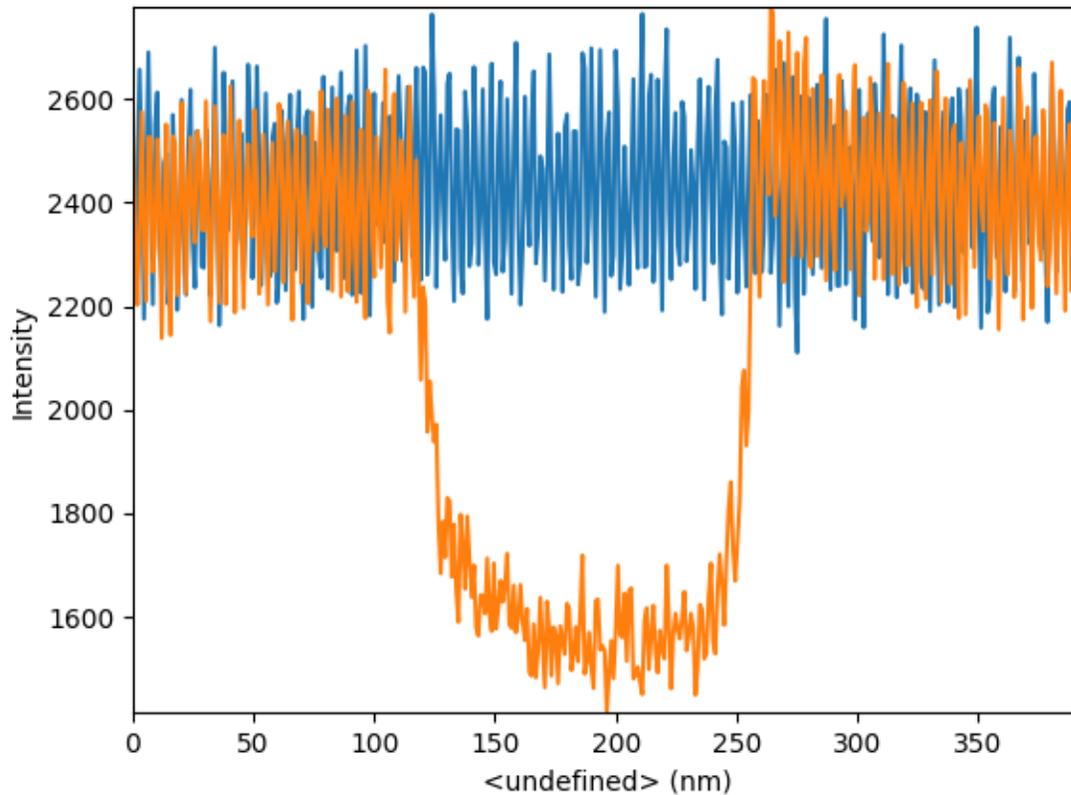
If you want to later update the ROI initialization with the modified parameters, you can print these:

```
print(tuple(line_roi))
```

```
(400.0, 250.0, 220.0, 600.0, 5.0)
```

You can now directly access the data of the profile objects, e.g. to plot both profiles in a single plot:

```
hs.plot.plot_spectra([profile1, profile2])
# Choose the fourth figure as gallery thumbnail:
# sphinx_gallery_thumbnail_number = 4
```



```
<Axes: xlabel='<undefined> (nm)', ylabel='Intensity'>
```

Since the profile is a signal object, you can use any other functionality provided by hyperspy, e.g. to save a profile as *.msa* text file:

```
profile1.save('extracted-line-profile.msa', format='XY')
```

Total running time of the script: (0 minutes 2.018 seconds)

22.5.4 Interactive integration of one dimensional signal

This example shows how to integrate a signal using an interactive ROI.

```
import hyperspy.api as hs
```

Create a signal:

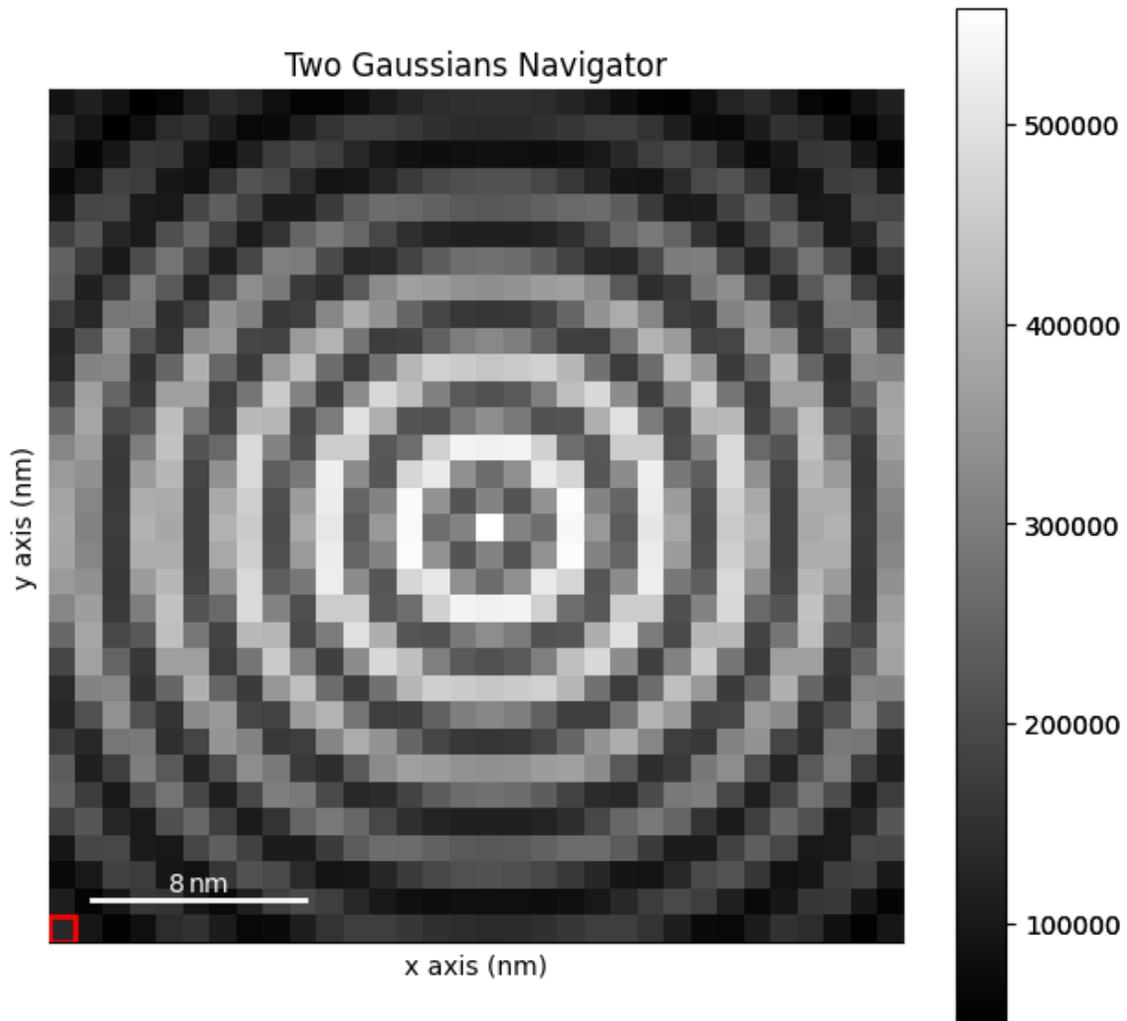
```
s = hs.data.two_gaussians()
```

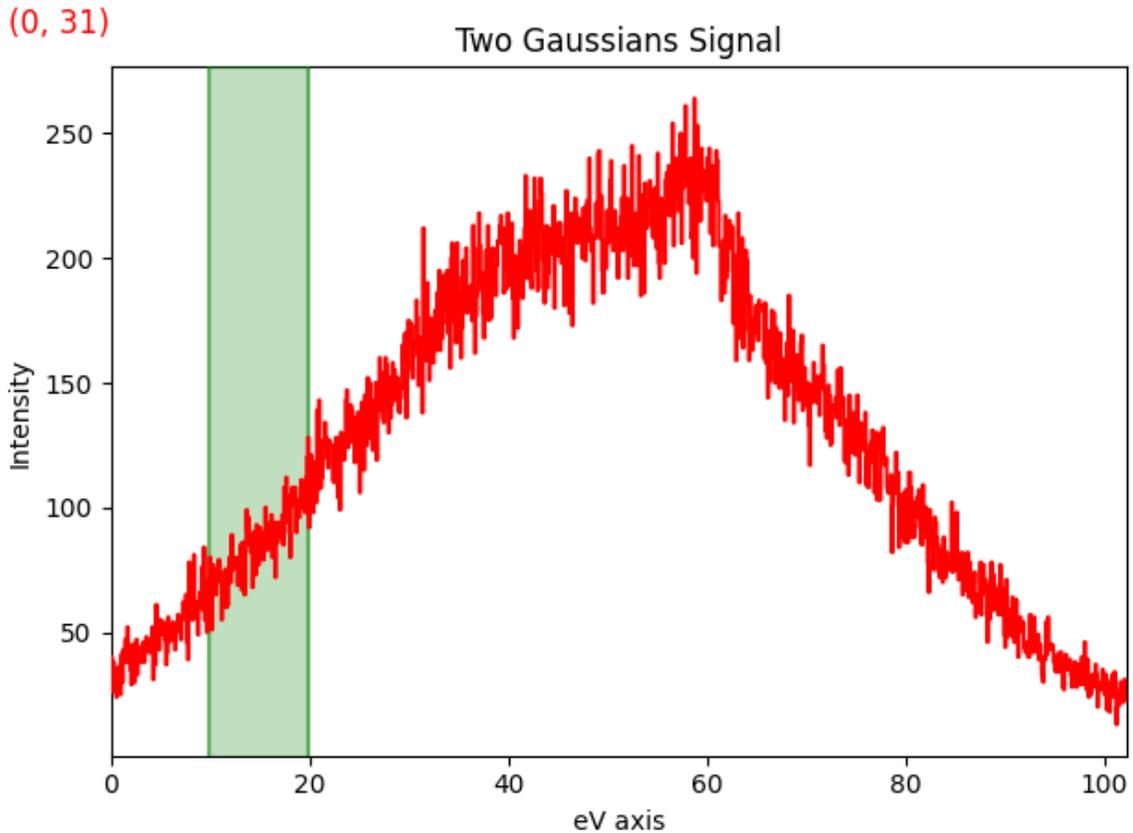
Create SpanROI:

```
roi = hs.roi.SpanROI(left=10, right=20)
```

Slice signal with roi with the ROI. By using the *interactive* function, the output signal `s_roi` will update automatically. The ROI will be added automatically on the signal figure:

```
s.plot()
sliced_signal = roi.interactive(s, axes=s.axes_manager.signal_axes)
# Choose the second figure as gallery thumbnail:
# sphinx_gallery_thumbnail_number = 2
```





Create a placeholder signal for the integrated signal and set metadata:

```
integrated_sliced_signal = sliced_signal.sum(axis=-1).T
integrated_sliced_signal.metadata.General.title = "Integrated intensity"
```

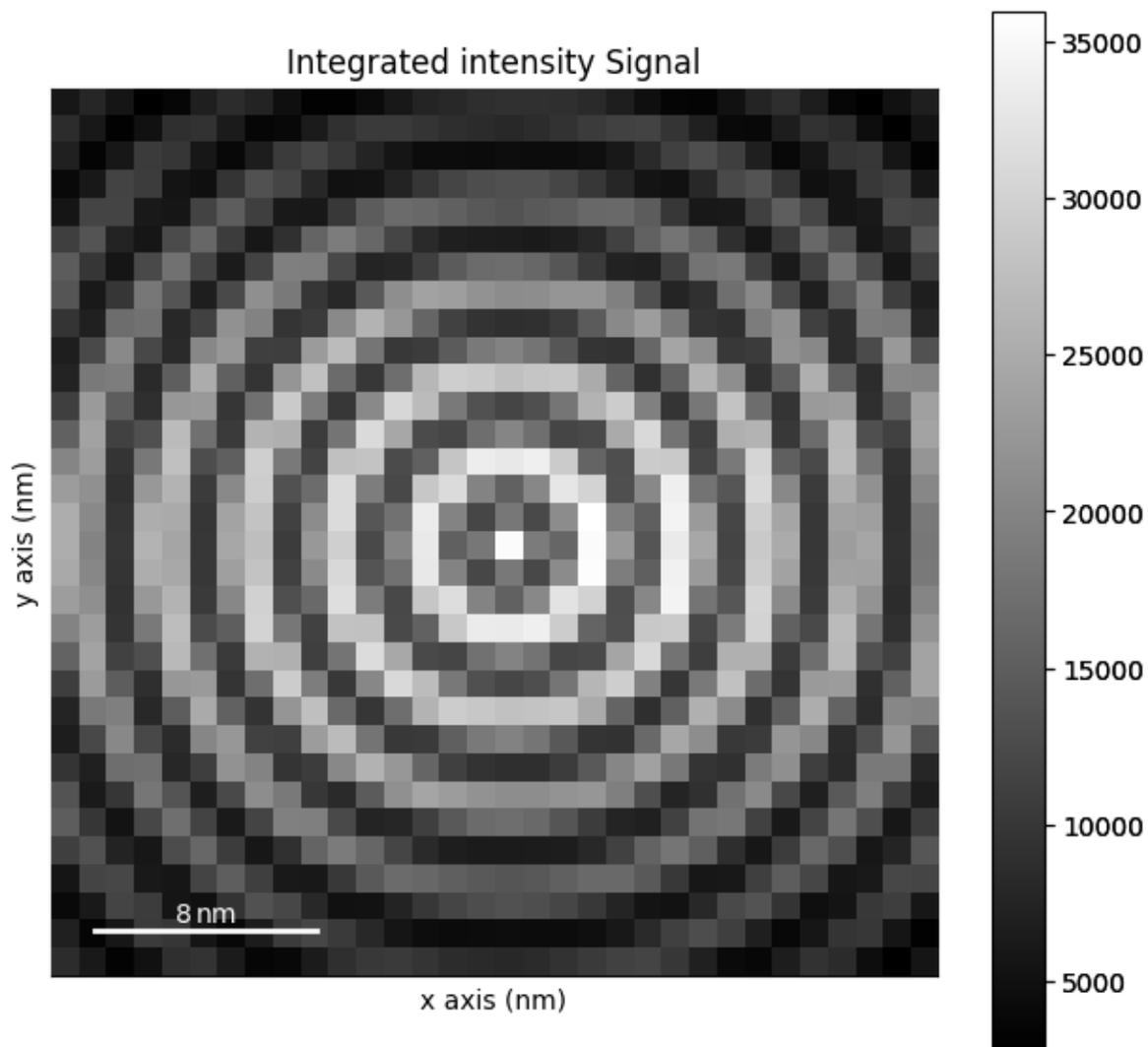
Create the interactive computation, which will update when the ROI `roi` is changed. We use the `out` argument to place the results of the integration in the placeholder signal defined in the previous step:

```
hs.interactive(
    sliced_signal.sum,
    axis=sliced_signal.axes_manager.signal_axes,
    event=roi.events.changed,
    recompute_out_event=None,
    out=integrated_sliced_signal,
)
```

```
<Signal2D, title: Integrated intensity, dimensions: (|32, 32)>
```

Plot the integrated sum signal:

```
integrated_sliced_signal.plot()
```



Total running time of the script: (0 minutes 1.681 seconds)

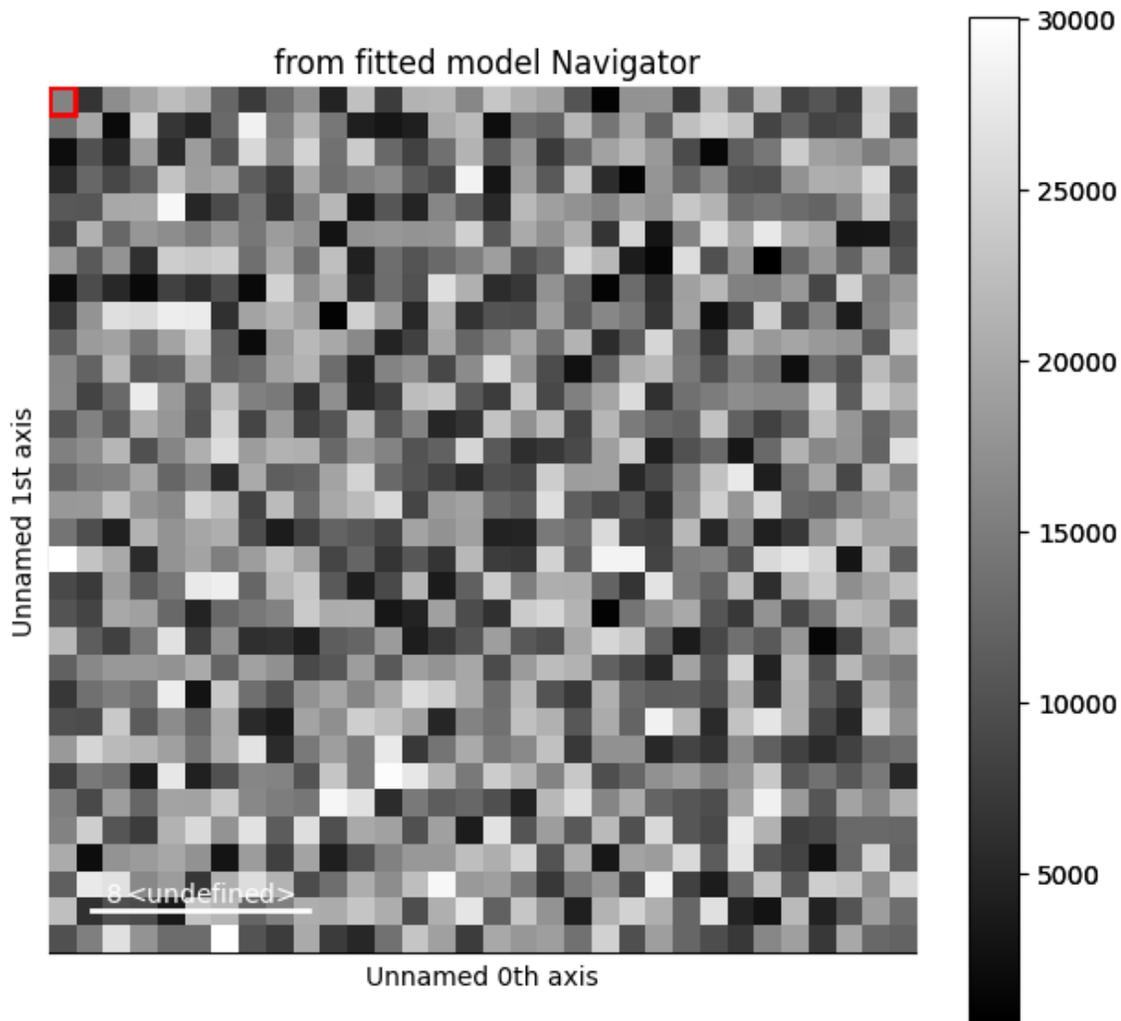
22.6 Simple simulations

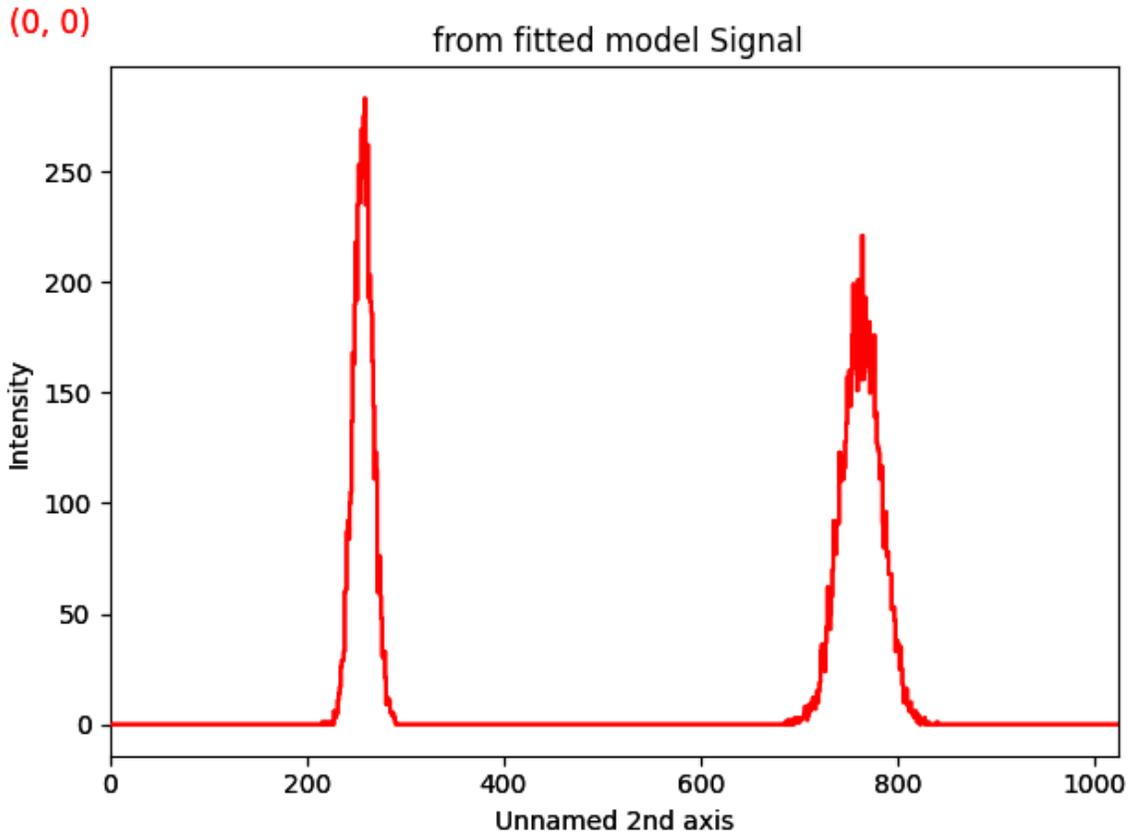
Below is a gallery of examples on simulating signals which can be used to test HyperSpy functionalities

22.6.1 Simple simulation (2 Gaussians)

Creates a 2D hyperspectrum consisting of two Gaussians and plots it.

This example can serve as starting point to test other functionalities on the simulated hyperspectrum.





```
Exception ignored in: <function tqdm.__del__ at 0x7f3e1ecc37e0>
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/hyperspy/envs/latest/lib/
↳python3.11/site-packages/tqdm/std.py", line 1148, in __del__
    self.close()
  File "/home/docs/checkouts/readthedocs.org/user_builds/hyperspy/envs/latest/lib/
↳python3.11/site-packages/tqdm/notebook.py", line 279, in close
    self.disp(bar_style='danger', check_delay=False)
    ^^^^^^^^^^^
AttributeError: 'tqdm_notebook' object has no attribute 'disp'
```

```
0%|          | 0/1024 [00:00<?, ?it/s]
21%|         | 217/1024 [00:00<00:00, 2166.17it/s]
43%|        | 443/1024 [00:00<00:00, 2219.01it/s]
65%|       | 669/1024 [00:00<00:00, 2234.82it/s]
87%|      | 893/1024 [00:00<00:00, 2226.70it/s]
100%|     | 1024/1024 [00:00<00:00, 2216.23it/s]
```

```

import numpy as np
import hyperspy.api as hs
import matplotlib.pyplot as plt

# Create an empty spectrum
s = hs.signals.Signal1D(np.zeros((32, 32, 1024)))

# Generate some simple data: two Gaussians with random centers and area

# First we create a model
m = s.create_model()

# Define the first gaussian
gs1 = hs.model.components1D.Gaussian()
# Add it to the model
m.append(gs1)

# Set the parameters
gs1.sigma.value = 10
# Make the center vary in the -5,5 range around 128
gs1.centre.map['values'][:] = 256 + (np.random.random((32, 32)) - 0.5) * 10
gs1.centre.map['is_set'][:] = True

# Make the area vary between 0 and 10000
gs1.A.map['values'][:] = 10000 * np.random.random((32, 32))
gs1.A.map['is_set'][:] = True

# Second gaussian
gs2 = hs.model.components1D.Gaussian()
# Add it to the model
m.append(gs2)

# Set the parameters
gs2.sigma.value = 20

# Make the center vary in the -10,10 range around 768
gs2.centre.map['values'][:] = 768 + (np.random.random((32, 32)) - 0.5) * 20
gs2.centre.map['is_set'][:] = True

# Make the area vary between 0 and 20000
gs2.A.map['values'][:] = 20000 * np.random.random((32, 32))
gs2.A.map['is_set'][:] = True

# Create the dataset
s_model = m.as_signal()

# Add noise
s_model.set_signal_origin("simulation")
s_model.add_poissonian_noise()

# Plot the result
s_model.plot()

```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 1.291 seconds)

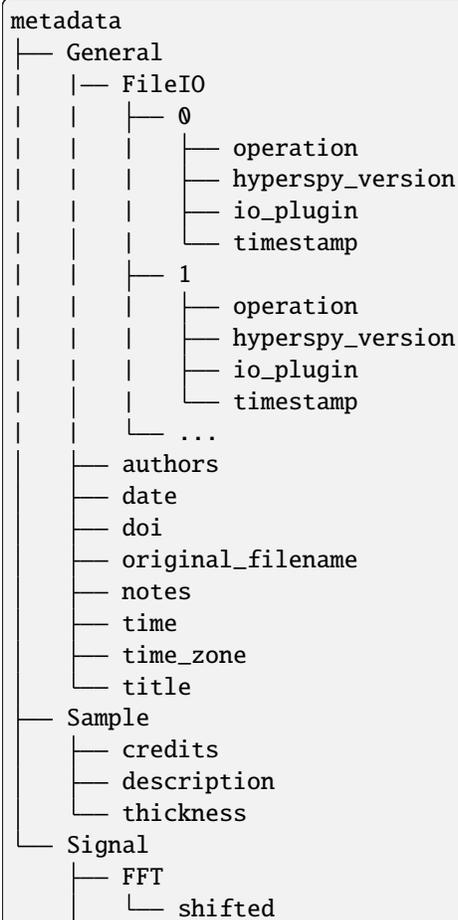
REFERENCE

23.1 Metadata structure

The *BaseSignal* class stores metadata in the *metadata* attribute, which has a tree structure. By convention, the node labels are capitalized and the leaves are not capitalized.

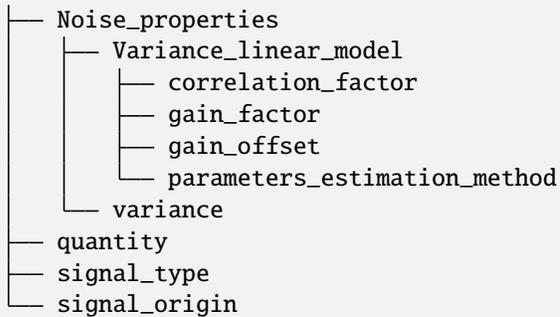
When a leaf contains a quantity that is not dimensionless, the units can be given in an extra leaf with the same label followed by the “_units” suffix. For example, an “energy” leaf should be accompanied by an “energy_units” leaf.

The metadata structure is represented in the following tree diagram. The default units are given in parentheses. Details about the leaves can be found in the following sections of this chapter.



(continues on next page)

(continued from previous page)



23.1.1 General

title

type: Str

A title for the signal, e.g. “Sample overview”

original_filename

type: Str

If the signal was loaded from a file this key stores the name of the original file.

time_zone

type: Str

The time zone as supported by the python-dateutil library, e.g. “UTC”, “Europe/London”, etc. It can also be a time offset, e.g. “+03:00” or “-05:00”.

time

type: Str

The acquisition or creation time in ISO 8601 time format, e.g. ‘13:29:10’.

date

type: Str

The acquisition or creation date in ISO 8601 date format, e.g. ‘2018-01-28’.

authors

type: Str

The authors of the data, in Latex format: Surname1, Name1 and Surname2, Name2, etc.

doi

type: Str

Digital object identifier of the data, e. g. doi:10.5281/zenodo.58841.

notes

type: Str

Notes about the data.

FileIO

Contains information about the software packages and versions used any time the Signal was created by reading the original data format (added in HyperSpy v1.7) or saved by one of HyperSpy's IO tools. If the signal is saved to one of the `hspy`, `zspy` or `nxs` formats, the metadata within the `FileIO` node will represent a history of the software configurations used when the conversion was made from the proprietary/original format to HyperSpy's format, as well as any time the signal was subsequently loaded from and saved to disk. Under the `FileIO` node will be one or more nodes named `0`, `1`, `2`, etc., each with the following structure:

operation

type: Str

This value will be either "load" or "save" to indicate whether this node represents a load from, or save to disk operation, respectively.

hyperspy_version

type: Str

The version number of the HyperSpy software used to extract a Signal from this data file or save this Signal to disk

io_plugin

type: Str

The specific input/output plugin used to originally extract this data file into a HyperSpy Signal or save it to disk – will be of the form `rscio.<plugin_name>`.

timestamp

type: Str

The timestamp of the computer running the data loading/saving process (in a timezone-aware format). The timestamp will be in ISO 8601 format, as produced by the `datetime.date.isoformat()`.

23.1.2 Sample

credits

type: Str

Acknowledgment of sample supplier, e.g. Prepared by Putin, Vladimir V.

description

type: Str

A brief description of the sample

thickness

type: Float

The thickness of the sample in m.

23.1.3 Signal

signal_type

type: Str

A term that describes the signal type, e.g. EDS, PES... This information can be used by HyperSpy to load the file as a specific signal class and therefore the naming should be standardised. Currently, HyperSpy provides special signal class for photoemission spectroscopy, electron energy loss spectroscopy and energy dispersive spectroscopy. The `signal_type` in these cases should be respectively PES, EELS and EDS_TEM (EDS_SEM).

signal_origin

type: Str

Describes the origin of the signal e.g. 'simulation' or 'experiment'.

record_by

Deprecated since version 1.2.

type: Str

One of 'spectrum' or 'image'. It describes how the data is stored in memory. If 'spectrum', the spectral data is stored in the faster index.

quantity

type: Str

The name of the quantity of the "intensity axis" with the units in round brackets if required, for example Temperature (K).

FFT

shifted

type: bool.

Specify if the FFT has the zero-frequency component shifted to the center of the signal.

Noise_properties

variance

type: float or BaseSignal instance.

The variance of the data. It can be a float when the noise is Gaussian or a *BaseSignal* instance if the noise is heteroscedastic, in which case it must have the same dimensions as *data*.

Variance_linear_model

In some cases the variance can be calculated from the data using a simple linear model: $\text{variance} = (\text{gain_factor} * \text{data} + \text{gain_offset}) * \text{correlation_factor}$.

gain_factor

type: Float

gain_offset

type: Float

correlation_factor

type: Float

parameters_estimation_method

type: Str

23.1.4 _Internal_parameters

This node is “private” and therefore is not displayed when printing the *metadata* attribute.

Stacking_history

Generated when using *stack()*. Used by *split()*, to retrieve the former list of signal.

step_sizes

type: list of int

Step sizes used that can be used in split.

axis

type: int

The axis index in axes manager on which the dataset were stacked.

Folding

Constains parameters that related to the folding/unfolding of signals.

23.1.5 Functions to handle the metadata

Existing nodes can be directly read out or set by adding the path in the metadata tree:

```
s.metadata.General.title = 'FlyingCircus'
s.metadata.General.title
```

The following functions can operate on the metadata tree. An example with the same functionality as the above would be:

```
s.metadata.set_item('General.title', 'FlyingCircus')
s.metadata.get_item('General.title')
```

Adding items***set_item()***

Given a path and value, easily set metadata items, creating any necessary nodes on the way.

add_dictionary()

Add new items from a given dictionary.

Output metadata

`get_item()`

Given an `item_path`, return the value of the metadata item.

`as_dictionary()`

Returns a dictionary representation of the metadata tree.

`export()`

Saves the metadata tree in pretty tree printing format in a text file. Takes `filename` as parameter.

Searching for keys

`has_item()`

Given an `item_path`, returns True if the item exists anywhere in the metadata tree.

Using the option `full_path=False`, the functions `has_item()` and `get_item()` can also find items by their key in the metadata when the exact path is not known. By default, only an exact match of the search string with the item key counts. The additional setting `wild=True` allows to search for a case-insensitive substring of the item key. The search functionality also accepts item keys preceded by one or several nodes of the path (separated by the usual full stop).

`has_item()`

For `full_path=False`, given a `item_key`, returns True if the item exists anywhere in the metadata tree.

`has_item()`

For `full_path=False`, `return_path=True`, returns the path or list of paths to any matching item(s).

`get_item()`

For `full_path=False`, returns the value or list of values for any matching item(s). Setting `return_path=True`, a tuple (value, path) is returned – or lists of tuples for multiple occurrences.

23.2 hyperspy.api

<code>hyperspy.api.get_configuration_directory_path()</code>	Return configuration path
<code>hyperspy.api.interactive(f[, event, ...])</code>	Chainable operations on Signals that update on events.
<code>hyperspy.api.load([filenames, signal_type, ...])</code>	Load potentially multiple supported files into HyperSpy.
<code>hyperspy.api.print_known_signal_types()</code>	Print all known <code>signal_types</code>
<code>hyperspy.api.set_log_level(level)</code>	Convenience function to set the log level of all hyperspy modules.
<code>hyperspy.api.stack(signal_list[, axis, ...])</code>	Concatenate the signals in the list over a given axis or a new axis.
<code>hyperspy.api.transpose(*args[, signal_axes, ...])</code>	Transposes all passed signals according to the specified options.

All public packages, functions and classes are available in this module.

When starting HyperSpy using the `hyperspy` script (e.g. by executing `hyperspy` in a console, using the context menu entries or using the links in the Start Menu, the `api` package is imported in the user namespace as `hs`, i.e. by executing the following:

```
>>> import hyperspy.api as hs
```

(Note that code snippets are indicated by three greater-than signs)

We recommend to import the HyperSpy API as above also when doing it manually. The docstring examples assume that `hyperspy.api` has been imported as `hs`, `numpy` as `np` and `matplotlib.pyplot` as `plt`.

Functions:

`get_configuration_directory_path()`

Return the configuration directory path.

`interactive()`

Define operations that are automatically recomputed on event changes.

`load()`

Load data into `BaseSignal` instances from supported files.

`preferences`

Preferences class instance to configure the default value of different parameters. It has a CLI and a GUI that can be started by executing its `gui` method i.e. `preferences.gui()`.

`print_known_signal_types()`

Print all known `signal_type`.

`set_log_level()`

Convenience function to set HyperSpy's the log level.

`stack()`

Stack several signals.

`transpose()`

Transpose a signal.

The `api` package contains the following submodules/packages:

`signals`

`Signal` classes which are the core of HyperSpy. Use this modules to create `Signal` instances manually from numpy arrays. Note that to load data from supported file formats is more convenient to use the `load` function.

`model`

Components that can be used to create a model for curve fitting.

`plot`

Plotting functions that operate on multiple signals.

`data`

Synthetic datasets.

`roi`

Region of interests (ROIs) that operate on `BaseSignal` instances and include widgets for interactive operation.

`samfire`

SAMFire utilities (strategies, Pool, fit convergence tests)

For more details see their docstrings.

`hyperspy.api.get_configuration_directory_path()`

Return configuration path

`hyperspy.api.interactive(f, event='auto', recompute_out_event='auto', *args, **kwargs)`

Chainable operations on Signals that update on events. The operation result will be updated when a given event is triggered.

Parameters

f

[[callable\(\)](#)] A function that returns an object and that optionally can place the result in an object given through the `out` keyword.

event

[(list of) [Event](#), [str](#) ("auto") or [None](#)] Update the result of the operation when the event is triggered. If "auto" and `f` is a method of a `Signal` class instance its `data_changed` event is selected if the function takes an `out` argument. If `None`, `update` is not connected to any event. The default is "auto". It is also possible to pass an iterable of events, in which case all the events are connected.

recompute_out_event

[(list of) [Event](#), [str](#) ("auto") or [None](#)] Optional argument. If supplied, this event causes a full recomputation of a new object. Both the data and axes of the new object are then copied over to the existing `out` object. Only useful for signals or other objects that have an attribute `axes_manager`. If "auto" and `f` is a method of a `Signal` class instance its `AxesManager` `any_axis_changed` event is selected. Otherwise, the signal `data_changed` event is selected. If `None`, `recompute_out` is not connected to any event. The default is "auto". It is also possible to pass an iterable of events, in which case all the events are connected.

***args**

Arguments to be passed to `f`.

****kwargs**

[[dict](#)] Keyword arguments to be passed to `f`.

Returns

[BaseSignal](#) or subclass

Signal updated with the operation result when a given event is triggered.

```
hyperspy.api.load(filenamees=None, signal_type=None, stack=False, stack_axis=None,
                  new_axis_name='stack_element', lazy=False, convert_units=False,
                  escape_square_brackets=False, stack_metadata=True, load_original_metadata=True,
                  show_progressbar=None, **kws)
```

Load potentially multiple supported files into HyperSpy.

Supported formats: hspy (HDF5), msa, Gatan dm3, Ripple (rpl+raw), Bruker bcf and spx, FEI ser and emi, SEMPER unf, EMD, EDAX spd/spc, CEOS prz tif, and a number of image formats.

Depending on the number of datasets to load in the file, this function will return a HyperSpy signal instance or list of HyperSpy signal instances.

Any extra keywords are passed to the corresponding reader. For available options, see their individual documentation.

Parameters

filenames

[[None](#), (list of) [str](#) or (list of) [pathlib.Path](#), default [None](#)] The filename to be loaded. If `None`, a window will open to select a file to load. If a valid filename is passed, that single file is loaded. If multiple file names are passed in a list, a list of objects or a single object containing multiple datasets, a list of signals or a stack of signals is returned. This behaviour is controlled by the `stack` parameter (see below). Multiple files can be loaded by using simple shell-style wildcards, e.g. `'my_file*.msa'` loads all the files that start by `'my_file'` and have the `'msa'` extension. Alternatively, regular expression type character classes can be used (e.g. `[a-z]` matches lowercase letters). See also the `escape_square_brackets` parameter.

signal_type

[`None`, `str`, default `None`] The acronym that identifies the signal type. May be any signal type provided by HyperSpy or by installed extensions as listed by `hs.print_known_signal_types()`. The value provided may determines the Signal subclass assigned to the data. If `None` (default), the value is read/guessed from the file. Any other value would override the value potentially stored in the file. For example, for electron energy-loss spectroscopy use ‘EELS’. If ‘’ (empty string) the value is not read from the file and is considered undefined.

stack

[`bool`, default `False`] Default `False`. If `True` and multiple filenames are passed, stacking all the data into a single object is attempted. All files must match in shape. If each file contains multiple (`N`) signals, `N` stacks will be created, with the requirement that each file contains the same number of signals.

stack_axis

[`None`, `int` or `str`, default `None`] If `None` (default), the signals are stacked over a new axis. The data must have the same dimensions. Otherwise, the signals are stacked over the axis given by its integer index or its name. The data must have the same shape, except in the dimension corresponding to *axis*.

new_axis_name

[`str`, optional] The name of the new axis (default ‘stack_element’), when *axis* is `None`. If an axis with this name already exists, it automatically appends ‘-i’, where *i* are integers, until it finds a name that is not yet in use.

lazy

[`bool`, default `False`] Open the data lazily - i.e. without actually reading the data from the disk until required. Allows opening arbitrary-sized datasets.

convert_units

[`bool`, default `False`] If `True`, convert the units using the `convert_to_units` method of the *axes_manager*. If `False`, does nothing.

escape_square_brackets

[`bool`, default `False`] If `True`, and *filenames* is a `str` containing square brackets, then square brackets are escaped before wildcard matching with `glob.glob()`. If `False`, square brackets are used to represent character classes (e.g. `[a-z]` matches lowercase letters).

stack_metadata

[`bool`, `int`] If integer, this value defines the index of the signal in the signal list, from which the metadata and `original_metadata` are taken. If `True`, the `original_metadata` and metadata of each signals are stacked and saved in `original_metadata.stack_elements` of the returned signal. In this case, the metadata are copied from the first signal in the list. If `False`, the metadata and `original_metadata` are not copied.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used. Only used with `stack=True`.

load_original_metadata

[`bool`, default `True`] If `True`, all metadata contained in the input file will be added to `original_metadata`. This does not affect parsing the metadata to metadata.

reader

[`None`, `str`, module, optional] Specify the file reader to use when loading the file(s). If `None` (default), will use the file extension to infer the file type and appropriate reader. If `str`, will select the appropriate file reader from the list of available readers in HyperSpy. If module, it

must implement the `file_reader` function, which returns a dictionary containing the data and metadata for conversion to a HyperSpy signal.

print_info: bool, optional

For SEMPER unf- and EMD (Berkeley)-files. If True, additional information read during loading is printed for a quick overview. Default False.

downsample

[`int` (1–4095), optional] For Bruker bcf files, if set to integer (≥ 2) (default 1), bcf is parsed into down-sampled size array by given integer factor, multiple values from original bcf pixels are summed forming downsampled pixel. This allows to improve signal and conserve the memory with the cost of lower resolution.

cutoff_at_kV

[`None`, `int`, `float`, optional] For Bruker bcf files and Jeol, if set to numerical (default is `None`), hypermap is parsed into array with depth cutoff at set energy value. This allows to conserve the memory by cutting-off unused spectral tails, or force enlargement of the spectra size. Bruker bcf reader accepts additional values for semi-automatic cutoff. “zealous” value truncates to the last non zero channel (this option should not be used for stacks, as low beam current EDS can have different last non zero channel per slice). “auto” truncates channels to SEM/TEM acceleration voltage or energy at last channel, depending which is smaller. In case the hv info is not there or hv is off (0 kV) then it fallbacks to full channel range.

select_type

[`'spectrum_image'`, `'image'`, `'single_spectrum'`, `None`, optional] If `None` (default), all data are loaded. For Bruker bcf and Velox emd files: if one of `'spectrum_image'`, `'image'` or `'single_spectrum'`, the loader returns either only the spectrum image, only the images (including EDS map for Velox emd files), or only the single spectra (for Velox emd files).

first_frame

[`int`, optional] Only for Velox emd files: load only the data acquired after the specified fname. Default 0.

last_frame

[`None`, `int`, optional] Only for Velox emd files: load only the data acquired up to specified fname. If `None` (default), load the data up to the end.

sum_frames

[`bool`, optional] Only for Velox emd files: if False, load each EDS frame individually. Default is True.

sum_EDS_detectors

[`bool`, optional] Only for Velox emd files: if True (default), the signals from the different detectors are summed. If False, a distinct signal is returned for each EDS detectors.

rebin_energy

[`int`, optional] Only for Velox emd files: rebin the energy axis by the integer provided during loading in order to save memory space. Needs to be a multiple of the length of the energy dimension (default 1).

SI_dtype

[`numpy.dtype`, `None`, optional] Only for Velox emd files: set the dtype of the spectrum image data in order to save memory space. If `None`, the default dtype from the Velox emd file is used.

load_SI_image_stack

[`bool`, optional] Only for Velox emd files: if True, load the stack of STEM images acquired simultaneously as the EDS spectrum image. Default is False.

dataset_path

[None, str, list of str, optional] For filetypes which support several datasets in the same file, this will only load the specified dataset. Several datasets can be loaded by using a list of strings. Only for EMD (NCEM) and hdf5 (USID) files.

stack_group

[bool, optional] Only for EMD NCEM. Stack datasets of groups with common name. Relevant for emd file version ≥ 0.5 where groups can be named 'group0000', 'group0001', etc.

ignore_non_linear_dims

[bool, optional] Only for HDF5 USID files: if True (default), parameters that were varied non-linearly in the desired dataset will result in Exceptions. Else, all such non-linearly varied parameters will be treated as linearly varied parameters and a Signal object will be generated.

only_valid_data

[bool, optional] Only for FEI emi/ser files in case of series or linescan with the acquisition stopped before the end: if True, load only the acquired data. If False, fill empty data with zeros. Default is False and this default value will change to True in version 2.0.

Returns

(list of) *BaseSignal* or subclass

Examples

Loading a single file providing the signal type:

```
>>> d = hs.load('file.dm3', signal_type="EDS_TEM")
```

Loading multiple files:

```
>>> d = hs.load(['file1.hspy', 'file2.hspy'])
```

Loading multiple files matching the pattern:

```
>>> d = hs.load('file*.hspy')
```

Loading multiple files containing square brackets in the filename:

```
>>> d = hs.load('file[*].hspy', escape_square_brackets=True)
```

Loading multiple files containing character classes (regular expression):

```
>>> d = hs.load('file[0-9].hspy')
```

Loading (potentially larger than the available memory) files lazily and stacking:

```
>>> s = hs.load('file*.blo', lazy=True, stack=True)
```

Specify the file reader to use

```
>>> s = hs.load('a_nexus_file.h5', reader='nxs')
```

Loading a file containing several datasets:

```
>>> s = hs.load("spameggsandham.nxs")
>>> s
[<Signal1D, title: spam, dimensions: (32,32|1024)>,
 <Signal1D, title: eggs, dimensions: (32,32|1024)>,
 <Signal1D, title: ham, dimensions: (32,32|1024)>]
```

Use list indexation to access single signal

```
>>> s[0]
<Signal1D, title: spam, dimensions: (32,32|1024)>
```

`hyperspy.api.print_known_signal_types()`

Print all known *signal_types*

This includes *signal_types* from all installed packages that extend HyperSpy.

Examples

```
>>> hs.print_known_signal_types()
+-----+-----+-----+-----+
| signal_type | aliases | class name | package |
+-----+-----+-----+-----+
| DielectricFunction | dielectric function | DielectricFunction | exspy |
| EDS_SEM | | EDSSEMSpectrum | exspy |
| EDS_TEM | | EDSTEMSpectrum | exspy |
| EELS | TEM EELS | EELSSpectrum | exspy |
| hologram | | HologramImage | holospy |
| MySignal | | MySignal | hspy_ext |
+-----+-----+-----+-----+
```

`hyperspy.api.set_log_level(level)`

Convenience function to set the log level of all hyperspy modules.

Note: The log level of all other modules are left untouched.

Parameters

level

[`int` or `str`] The log level to set. Any values that `logging.Logger.setLevel()` accepts are valid. The default options are:

- 'CRITICAL'
- 'ERROR'
- 'WARNING'
- 'INFO'
- 'DEBUG'
- 'NOTSET'

Examples

For normal logging of hyperspy functions, you can set the log level like this:

```
>>> import hyperspy.api as hs
>>> hs.set_log_level('INFO')
>>> hs.load('my_file.dm3')
INFO:rsciio.digital_micrograph:DM version: 3
INFO:rsciio.digital_micrograph:size 4796607 B
INFO:rsciio.digital_micrograph:Is file Little endian? True
INFO:rsciio.digital_micrograph:Total tags in root group: 15
<Signal2D, title: My file, dimensions: (|1024, 1024)>
```

If you need the log output during the initial import of hyperspy, you should set the log level like this:

```
>>> from hyperspy.logger import set_log_level
>>> hs.set_log_level('DEBUG')
>>> import hyperspy.api as hs
DEBUG:hyperspy.gui:Loading hyperspy.gui
DEBUG:hyperspy.gui:Current MPL backend: TkAgg
DEBUG:hyperspy.gui:Current ETS toolkit: qt4
DEBUG:hyperspy.gui:Current ETS toolkit set to: null
```

`hyperspy.api.stack(signal_list, axis=None, new_axis_name='stack_element', lazy=None, stack_metadata=True, show_progressbar=None, **kwargs)`

Concatenate the signals in the list over a given axis or a new axis.

The title is set to that of the first signal in the list.

Parameters

signal_list

[list of *BaseSignal*] List of signals to stack.

axis

[None, int or str] If None, the signals are stacked over a new axis. The data must have the same dimensions. Otherwise the signals are stacked over the axis given by its integer index or its name. The data must have the same shape, except in the dimension corresponding to *axis*. If the stacking axis of the first signal is uniform, it is extended up to the new length; if it is non-uniform, the axes vectors of all signals are concatenated along this direction; if it is a *FunctionalDataAxis*, it is extended based on the expression of the first signal (and its sub axis *x* is handled as above depending on whether it is uniform or not).

new_axis_name

[str] The name of the new axis when *axis* is None. If an axis with this name already exists it automatically append '-i', where *i* are integers, until it finds a name that is not yet in use.

lazy

[bool or None] Returns a LazySignal if True. If None, only returns lazy result if at least one is lazy.

stack_metadata

[{bool, int}] If integer, this value defines the index of the signal in the signal list, from which the metadata and `original_metadata` are taken. If True, the `original_metadata` and metadata of each signals are stacked and saved in `original_metadata.stack_elements` of the returned signal. In this case, the metadata are copied from the first signal in the list. If False, the metadata and `original_metadata` are not copied.

show_progressbar

[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

Returns

BaseSignal

Examples

```
>>> data = np.arange(20)
>>> s = hs.stack(
...     [hs.signals.Signal1D(data[:10]), hs.signals.Signal1D(data[10:])]
... )
>>> s
<Signal1D, title: Stack of , dimensions: (2|10)>
>>> s.data
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

`hyperspy.api.transpose(*args, signal_axes=None, navigation_axes=None, optimize=False)`

Transposes all passed signals according to the specified options.

For parameters see `BaseSignal.transpose`.

Examples

```
>>> signal_iterable = [
...     hs.signals.BaseSignal(np.random.random((2,)*(i+1))) for i in range(3)
... ]
>>> signal_iterable
[<BaseSignal, title: , dimensions: (|2)>,
 <BaseSignal, title: , dimensions: (|2, 2)>,
 <BaseSignal, title: , dimensions: (|2, 2, 2)>]
>>> hs.transpose(*signal_iterable, signal_axes=1)
[<Signal1D, title: , dimensions: (|2)>,
 <Signal1D, title: , dimensions: (2|2)>,
 <Signal1D, title: , dimensions: (2, 2|2)>]
```

23.3 hyperspy.api.data

The `hyperspy.api.data` module includes synthetic data signal.

`hyperspy.api.data.atomic_resolution_image()`

Get an artificial atomic resolution image.

Returns

Signal2D

Examples

```
>>> s = hs.data.atomic_resolution_image()
>>> s.plot()
```

`hyperspy.api.data.luminescence_signal` (*navigation_dimension=0, uniform=False, add_baseline=False, add_noise=True, random_state=None*)

Get an artificial luminescence signal in wavelength scale (nm, uniform) or energy scale (eV, non-uniform), simulating luminescence data recorded with a diffracting spectrometer. Some random noise is also added to the spectrum, to simulate experimental noise.

Parameters

navigation_dimension

[[int](#)] The navigation dimension(s) of the signal. 0 = single spectrum, 1 = linescan, 2 = spectral map etc...

uniform

[[bool](#)] return uniform (wavelength) or non-uniform (energy) spectrum

add_baseline

[[bool](#)] If true, adds a constant baseline to the spectrum. Conversion to energy representation will turn the constant baseline into inverse powerlaw.

add_noise

[[bool](#)] If True, add noise to the signal. See note to seed the noise to generate reproducible noise.

random_state

[[None](#), [int](#) or [numpy.random.Generator](#), default [None](#)] Random seed used to generate the data.

Returns

Signal1D

See also:

[atomic_resolution_image](#)

Examples

```
>>> import hyperspy.api as hs
>>> s = hs.data.luminescence_signal()
>>> s.plot()
```

With constant baseline

```
>>> s = hs.data.luminescence_signal(uniform=True, add_baseline=True)
>>> s.plot()
```

To make the noise the same for multiple spectra, which can be useful for testing fitting routines

```
>>> s1 = hs.data.luminescence_signal(random_state=10)
>>> s2 = hs.data.luminescence_signal(random_state=10)
>>> (s1.data == s2.data).all()
True
```

2D map

```
>>> s = hs.data.luminescence_signal(navigation_dimension=2)
>>> s.plot()
```

`hyperspy.api.data.two_gaussians`(*add_noise=True*, *return_model=False*)

Create synthetic data consisting of two Gaussian functions with random centers and area

Parameters

add_noise

[bool] If True, add noise to the signal.

return_model

[bool] If True, returns the model in addition to the signal.

Returns

BaseSignal or tuple

Returns tuple when `return_model=True`.

`hyperspy.api.data.wave_image`(*angle=45*, *wavelength=10*, *shape=(256, 256)*, *add_noise=True*, *random_state=None*)

Returns a wave image generated using the sinus function.

Parameters

angle

[float, optional] The angle in degree.

wavelength

[float, optional] The wavelength the wave in pixel. The default is 10

shape

[tuple of float, optional] The shape of the data. The default is (256, 256).

add_noise

[bool] If True, add noise to the signal. See note to seed the noise to generate reproducible noise.

random_state

[None, int or `numpy.random.Generator`, default None] Random seed used to generate the data.

Returns

Signal2D

23.4 hyperspy.api.model

Model functions.

The model module contains the following submodules:

hyperspy.api.model.components1D

1D components for HyperSpy model.

hyperspy.api.model.components2D

2D components for HyperSpy model.

<i>components1D</i>	Components that can be used to define a 1D model for e.g. curve fitting.
<i>components2D</i>	Components that can be used to define a 2D model for e.g. curve fitting.

23.4.1 components1D

Components that can be used to define a 1D model for e.g. curve fitting.

Writing a new template is easy: see the user guide documentation on creating components.

For more details see each component docstring.

Arctan

Arctan function component.

Bleasdale

Bleasdale function component.

Doniach

Doniach Sunjic lineshape component.

Erf

Error function component.

Exponential

Exponential function component.

Expression

Create a component from a string expression.

Gaussian

Normalized Gaussian function component.

GaussianHF

Normalized gaussian function component, with a `fwfm` parameter

HeavisideStep

The Heaviside step function.

Logistic

Logistic function (sigmoid or s-shaped curve) component.

Lorentzian

Cauchy-Lorentz distribution (a.k.a. Lorentzian function) component.

Offset

Component to add a constant value in the y-axis.

Polynomial

n-order polynomial component.

PowerLaw

Power law component.

RC

ScalableFixedPattern

Fixed pattern component with interpolation support.

SkewNormal

Skew normal distribution component.

SplitVoigt

Split pseudo-Voigt component.

Voigt

Voigt component.

```
class hyperspy.api.model.components1D.Arctan(A=1.0, k=1.0, x0=1.0, module=['numpy', 'scipy'],
                                             **kwargs)
```

Bases: *Expression*

Arctan function component.

$$f(x) = A \cdot \arctan[k(x - x_0)]$$

Variable	Parameter
A	A
k	k
x_0	x0

Parameters

A

[float] Amplitude parameter. $\lim_{x \rightarrow -\infty} f(x) = -A$ and $\lim_{x \rightarrow \infty} f(x) = A$

k

[float] Slope (steepness of the step). The larger k , the sharper the step.

x0

[float] Center parameter (position of zero crossing $f(x_0) = 0$).

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

```
class hyperspy.api.model.components1D.Bleasdale(a=1.0, b=1.0, c=1.0, module=None, **kwargs)
```

Bases: *Expression*

Bleasdale function component.

Also called the Bleasdale-Nelder function. Originates from the description of the yield-density relationship in crop growth.

$$f(x) = (a + b \cdot x)^{-1/c}$$

Parameters

a

[float, default=1.0] The value of Parameter a.

b

[float, default=1.0] The value of Parameter b.

c
 [float, default=1.0] The value of Parameter c.

****kwargs**
 Extra keyword arguments are passed to *Expression*.

Notes

For $(a + b \cdot x) \leq 0$, the component will be set to 0.

Parameters

parameter_name_list
 [list] The list of parameter names.

linear_parameter_list
 [list, optional] The list of linear parameter. The default is None.

grad_a(x)
 Returns d(function)/d(parameter_1)

grad_b(x)
 Returns d(function)/d(parameter_1)

grad_c(x)
 Returns d(function)/d(parameter_1)

class hyperspy.api.model.components1D.**Doniach**(*centre=0.0, A=1.0, sigma=1.0, alpha=0.5, module=['numpy', 'scipy'], **kwargs*)

Bases: *Expression*

Doniach Sunjic lineshape component.

$$f(x) = \frac{A \cos\left[\frac{\pi\alpha}{2} + (1 - \alpha) \tan^{-1}\left(\frac{x - \text{centre} + dx}{\sigma}\right)\right]}{(\sigma^2 + (x - \text{centre} + dx)^2)^{\frac{(1-\alpha)}{2}}}$$

$$dx = \frac{2.354820\sigma}{2 \tan\left[\frac{\pi}{2-\alpha}\right]}$$

Variable	Parameter
<i>A</i>	A
σ	sigma
α	alpha
<i>centre</i>	centre

Parameters

A
 [float] Height

sigma
 [float] Variance parameter of the distribution

alpha
 [float] Tail or asymmetry parameter

centre

[float] Location of the maximum (peak position).

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Notes

This is an asymmetric lineshape, originally design for xps but generally useful for fitting peaks with low side tails See Doniach S. and Sunjic M., J. Phys. 4C31, 285 (1970) or http://www.casaxps.com/help_manual/line_shapes.htm for a more detailed description

Parameters**parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal, x1, x2, only_current=False*)

Estimate the Donach by calculating the median (centre) and the variance parameter (sigma).

Note that an insufficient range will affect the accuracy of this method and that this method doesn't estimate the asymmetry parameter (alpha).

Parameters**signal**

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns**bool**

Returns True when the parameters estimation is successful

Examples

```
>>> g = hs.model.components1D.Lorentzian()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:, :] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True
```

```
class hyperspy.api.model.components1D.Erf(A=1.0, sigma=1.0, origin=0.0, module=['numpy', 'scipy'],
    **kwargs)
```

Bases: *Expression*

Error function component.

$$f(x) = \frac{A}{2} \operatorname{erf} \left[\frac{(x - x_0)}{\sqrt{2}\sigma} \right]$$

Variable	Parameter
A	A
σ	sigma
x_0	origin

Parameters

A

[float] The min/max values of the distribution are $-A/2$ and $A/2$.

sigma

[float] Width of the distribution.

origin

[float] Position of the zero crossing.

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

```
class hyperspy.api.model.components1D.Exponential(A=1.0, tau=1.0, module=None, **kwargs)
```

Bases: *Expression*

Exponential function component.

$$f(x) = A \cdot \exp \left(-\frac{x}{\tau} \right)$$

Variable	Parameter
A	A
τ	tau

Parameters

A: float

Maximum intensity

tau: float

Scale parameter (time constant)

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Parameters**parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal, x1, x2, only_current=False*)

Estimate the parameters for the exponential component by splitting the signal window into two regions and using their geometric means

Parameters**signal**

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns

bool

```
class hyperspy.api.model.components1D.Expression(expression, name, position=None, module='numpy',
                                                autodoc=True, add_rotation=False,
                                                rotation_center=None, rename_pars={},
                                                compute_gradients=True,
                                                linear_parameter_list=None,
                                                check_parameter_linearity=True, **kwargs)
```

Bases: *Component*

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters**expression**

[str] Component function in SymPy text expression format with substitutions separated by `;`. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables *x* and *y* must be included for 1D or 2D components. Also, if *module* is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

name

[str] Name of the component.

position

[str, optional] The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“*x0*”, “*y0*”).

module

[`None` or `str` {"numpy" | "numexpr" | "scipy"}], default "numpy"] Module used to evaluate the function. `numexpr` is often faster but it supports fewer functions and requires installing `numexpr`. If `None`, the "numexpr" will be used if installed.

add_rotation

[`bool`, default `False`] This is only relevant for 2D components. If `True` it automatically adds `rotation_angle` parameter.

rotation_center

[`None` or `tuple`] If `None`, the rotation center is the center i.e. (0, 0) if `position` is not defined, otherwise the center is the coordinates specified by `position`. Alternatively a tuple with the (x, y) coordinates of the center can be provided.

rename_pars

[`dict`] The desired name of a parameter may sometimes coincide with e.g. the name of a scientific function, what prevents using it in the `expression`. `rename_parameters` is a dictionary to map the name of the parameter in the `expression`` to the desired name of the parameter in the `Component`. For example: {"_gamma": "gamma"}.

compute_gradients

[`bool`, optional] If `True`, compute the gradient automatically using `sympy`. If `sympy` does not support the calculation of the partial derivatives, for example in case of expression containing a "where" condition, it can be disabled by using `compute_gradients=False`.

linear_parameter_list

[`list`] A list of the components parameters that are known to be linear parameters.

check_parameter_linearity

[`bool`] If `True`, automatically check if each parameter is linear and set its corresponding attribute accordingly. If `False`, the default is to set all parameters, except for those who are specified in `linear_parameter_list`.

****kwargs**

[`dict`] Keyword arguments can be used to initialise the value of the parameters.

Notes

As of version 1.4, `Sympy`'s `lambdify` function, that the `Expression` components uses internally, does not support the differentiation of some expressions, for example those containing a "where" condition. In such cases, the gradients can be set manually if required.

Examples

The following creates a Gaussian component and set the initial value of the parameters:

```
>>> hs.model.components1D.Expression(
... expression="height * exp(-(x - x0) ** 2 * 4 * log(2)/ fwhm ** 2)",
... name="Gaussian",
... height=1,
... fwhm=1,
... x0=0,
... position="x0",)
<Gaussian (Expression component)>
```

Substitutions for long or complicated expressions are separated by semicolons:

```

>>> expr = 'A*B/(A+B) ; A = sin(x)+one; B = cos(y) - two; y = tan(x)'
>>> comp = hs.model.components1D.Expression(
...     expression=expr,
...     name='my function'
... )
>>> comp.parameters
(<Parameter one of my function component>,
 <Parameter two of my function component>)

```

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

compile_function(*module*, *position=False*)

Compile the function and calculate the gradient automatically when possible. Useful to recompile the function and gradient with a different module.

function_nd(*args)

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly to obtain the fitted component without iterating over the navigation axes.

class hyperspy.api.model.components1D.**Gaussian**(*A=1.0*, *sigma=1.0*, *centre=0.0*, *module=None*, **kwargs)

Bases: *Expression*

Normalized Gaussian function component.

$$f(x) = \frac{A}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right]$$

Variable	Parameter
A	A
σ	sigma
x_0	centre

Parameters

A

[float] Area, equals height scaled by $\sigma\sqrt{(2\pi)}$. GaussianHF implements the Gaussian function with a height parameter corresponding to the peak height.

sigma

[float] Scale parameter of the Gaussian distribution.

centre

[float] Location of the Gaussian maximum (peak position).

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

See also:

GaussianHF

Attributes

fwhm

[float] Convenience attribute to get and set the full width at half maximum.

height

[float] Convenience attribute to get and set the height.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal*, *x1*, *x2*, *only_current=False*)

Estimate the Gaussian by calculating the momenta.

Parameters

signal

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns

bool

Notes

Adapted from <https://scipy-cookbook.readthedocs.io/items/FittingData.html>

Examples

```

>>> g = hs.model.components1D.Gaussian()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:, :] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True

```

```
class hyperspy.api.model.components1D.GaussianHF(height=1.0, fwhm=1.0, centre=0.0, module=None,
**kwargs)
```

Bases: [Expression](#)

Normalized gaussian function component, with a `fwhm` parameter instead of the `sigma` parameter, and a `height` parameter instead of the area parameter `A` (scaling difference of $\sigma\sqrt{(2\pi)}$). This makes the parameter vs. peak maximum independent of σ , and thereby makes locking of the parameter more viable. As long as there is no binning, the `height` parameter corresponds directly to the peak maximum, if not, the value is scaled by a linear constant (`signal_axis.scale`).

$$f(x) = h \cdot \exp\left[-\frac{4 \log 2 (x - c)^2}{W^2}\right]$$

Variable	Parameter
h	height
W	fwhm
c	centre

Parameters

height: float

The height of the peak. If there is no binning, this corresponds directly to the maximum, otherwise the maximum divided by `signal_axis.scale`

fwhm: float

The full width half maximum value, i.e. the width of the gaussian at half the value of gaussian peak (at centre).

centre: float

Location of the gaussian maximum, also the mean position.

****kwargs**

Extra keyword arguments are passed to the [Expression](#) component.

See also:

[Gaussian](#)

Attributes

A

[float] Convenience attribute to get, set the area and defined for compatibility with *Gaussian* component.

sigma

[float] Convenience attribute to get, set the width and defined for compatibility with *Gaussian* component.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal*, *x1*, *x2*, *only_current=False*)

Estimate the gaussian by calculating the momenta.

Parameters

signal

[*Signal1D*]

x1

[*float*] Defines the left limit of the spectral range to use for the estimation.

x2

[*float*] Defines the right limit of the spectral range to use for the estimation.

only_current

[*bool*] If False estimates the parameters for the full dataset.

Returns

bool

Notes

Adapted from <https://scipy-cookbook.readthedocs.io/items/FittingData.html>

Examples

```
>>> g = hs.model.components1D.GaussianHF()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:, :] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True
```

integral_as_signal()

Utility function to get gaussian integral as *Signal1D*

class hyperspy.api.model.components1D.**HeavisideStep**(*A=1.0*, *n=0.0*, *module='numpy'*,
compute_gradients=True, ***kwargs*)

Bases: *Expression*

The Heaviside step function.

Based on the corresponding *numpy function* using the half maximum definition for the central point:

$$f(x) = \begin{cases} 0 & x < n \\ A/2 & x = n \\ A & x > n \end{cases}$$

Variable	Parameter
<i>n</i>	centre
<i>A</i>	height

Parameters

- n**
[float] Location parameter defining the x position of the step.
- A**
[float] Height parameter for $x > n$.
- **kwargs**
Extra keyword arguments are passed to the *Expression* component.

Parameters

- parameter_name_list**
[list] The list of parameter names.
- linear_parameter_list**
[list, optional] The list of linear parameter. The default is None.

```
class hyperspy.api.model.components1D.Logistic(a=1.0, b=1.0, c=1.0, origin=0.0, module=None,
                                              **kwargs)
```

Bases: *Expression*

Logistic function (sigmoid or s-shaped curve) component.

$$f(x) = \frac{a}{1 + b \cdot \exp[-c((x - x_0))]}$$

Variable	Parameter
A	a
b	b
c	c
x_0	origin

Parameters

- a**
[float] The curve's maximum y-value, $\lim_{x \rightarrow \infty} (y) = a$
- b**
[float] Additional parameter: $b > 1$ shifts origin to larger values; $0 < b < 1$ shifts origin to smaller values; $b < 0$ introduces an asymptote
- c**
[float] Logistic growth rate or steepness of the curve
- origin**
[float] Position of the sigmoid's midpoint
- **kwargs**
[dict] Extra keyword arguments are passed to the *Expression* component.

Parameters

- parameter_name_list**
[list] The list of parameter names.
- linear_parameter_list**
[list, optional] The list of linear parameter. The default is None.

```
class hyperspy.api.model.components1D.Lorentzian(A=1.0, gamma=1.0, centre=0.0, module=None,
**kwargs)
```

Bases: *Expression*

Cauchy-Lorentz distribution (a.k.a. Lorentzian function) component.

$$f(x) = \frac{A}{\pi} \left[\frac{\gamma}{(x - x_0)^2 + \gamma^2} \right]$$

Variable	Parameter
A	A
γ	gamma
x_0	centre

Parameters

A

[float] Area parameter, where $A/(\gamma\pi)$ is the maximum (height) of peak.

gamma

[float] Scale parameter corresponding to the half-width-at-half-maximum of the peak, which corresponds to the interquartile spread.

centre

[float] Location of the peak maximum.

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

For convenience the ``fwhm`` and ``height`` attributes can be used to get and set the full-width-half-maximum and height of the distribution, respectively.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

```
estimate_parameters(signal, x1, x2, only_current=False)
```

Estimate the Lorentzian by calculating the median (centre) and half the interquartile range (gamma).

Note that an insufficient range will affect the accuracy of this method.

Parameters

signal

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns**bool****Notes**Adapted from gaussian.py and https://en.wikipedia.org/wiki/Cauchy_distribution**Examples**

```

>>> g = hs.model.components1D.Lorentzian()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True

```

class hyperspy.api.model.components1D.**Offset**(*offset=0.0*)

Bases: *Component*

Component to add a constant value in the y-axis.

$$f(x) = k$$

Variable	Parameter
k	offset

Parameters**offset****[float]** The offset to be fitted**Parameters****parameter_name_list****[list]** The list of parameter names.**linear_parameter_list****[list, optional]** The list of linear parameter. The default is None.

estimate_parameters(*signal, x1, x2, only_current=False*)

Estimate the parameters by the two area method

Parameters**signal****[Signal1D]****x1****[float]** Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns**bool****function_nd**(*axis*)

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly to obtain the fitted component without iterating over the navigation axes.

class hyperspy.api.model.components1D.**Polynomial**(*order=2, module=None, **kwargs*)Bases: *Expression*

n-order polynomial component.

Polynomial component consisting of order + 1 parameters. The parameters are named “a” followed by the corresponding order, i.e.

$$f(x) = a_2x^2 + a_1x^1 + a_0$$

Zero padding is used for polynomial of order > 10.

Parameters**order**

[int] Order of the polynomial, must be different from 0.

****kwargs**Keyword arguments can be used to initialise the value of the parameters, i.e. a2=2, a1=3, a0=1. Extra keyword arguments are passed to the *Expression* component.**Parameters****parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal, x1, x2, only_current=False*)

Estimate the parameters by the two area method

Parameters**signal**

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns**bool**

```
class hyperspy.api.model.components1D.PowerLaw(A=1000000.0, r=3.0, origin=0.0, left_cutoff=0.0,  
                                              module=None, compute_gradients=False, **kwargs)
```

Bases: *Expression*

Power law component.

$$f(x) = A \cdot (x - x_0)^{-r}$$

Variable	Parameter
A	A
r	r
x_0	origin

Parameters

A

[float] Height parameter.

r

[float] Power law coefficient.

origin

[float] Location parameter.

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Attributes

left_cutoff

[float] For $x \leq \text{left_cutoff}$, the function returns 0. Default value is 0.0.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

```
estimate_parameters(signal, x1, x2, only_current=False, out=False)
```

Estimate the parameters for the power law component by the two area method.

Parameters

signal

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False, estimates the parameters for the full dataset.

out

[[bool](#)] If True, returns the result arrays directly without storing in the parameter maps/values. The returned order is (A, r).

Returns**bool**

Exit status required for the `remove_background()` function.

class `hyperspy.api.model.components1D.RC(Vmax=1.0, V0=0.0, tau=1.0, module=None, **kwargs)`

Bases: [Expression](#)

RC function component (based on the time-domain capacitor voltage response of an RC-circuit)

$$f(x) = V_0 + V_{\max} \left[1 - \exp\left(-\frac{x}{\tau}\right) \right]$$

Variable	Parameter
V_{\max}	Vmax
V_0	V0
τ	tau

Parameters**Vmax**

[[float](#)] maximum voltage, asymptote of the function for $\lim_{x \rightarrow \infty}$

V0

[[float](#)] vertical offset

tau

[[float](#)] tau=RC is the RC circuit time constant (voltage rise time)

****kwargs**

Extra keyword arguments are passed to the [Expression](#) component.

Parameters**parameter_name_list**

[[list](#)] The list of parameter names.

linear_parameter_list

[[list](#), optional] The list of linear parameter. The default is None.

class `hyperspy.api.model.components1D.ScalableFixedPattern(signalID, yscale=1.0, xscale=1.0, shift=0.0, interpolate=True)`

Bases: [Component](#)

Fixed pattern component with interpolation support.

$$f(x) = a \cdot s(b \cdot x - x_0) + c$$

Variable	Parameter
a	yscale
b	xscale
x_0	shift

Parameters**yscale**

[float] The scaling factor in y (intensity axis).

xscale

[float] The scaling factor in x.

shift

[float] The shift of the component

interpolate

[bool] If False no interpolation is performed and only a y-scaled spectrum is returned.

Examples

The fixed pattern is defined by a Signal1D of navigation 0 which must be provided to the ScalableFixedPattern constructor, e.g.:

```
>>> s = hs.load('data.hspy')
>>> my_fixed_pattern = hs.model.components1D.ScalableFixedPattern(s)
```

Attributes**yscale**

[Parameter] The scaling factor in y (intensity axis).

xscale

[Parameter] The scaling factor in x.

shift

[Parameter] The shift of the component

interpolate

[bool] If False no interpolation is performed and only a y-scaled spectrum is returned.

Methods

<code>prepare_interpolator(**kwargs)</code>	Fine-tune the interpolation.
---	------------------------------

Parameters**parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

function_nd(*axis*)

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly to obtain the fitted component without iterating over the navigation axes.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

prepare_interpolator(kwargs)**

Fine-tune the interpolation.

Parameters**x**

[array] The spectral axis of the fixed pattern

****kwargs**

[dict] Keywords argument are passed to `scipy.interpolate.make_interp_spline()`

```
class hyperspy.api.model.components1D.SkewNormal(x0=0.0, A=1.0, scale=1.0, shape=0.0,
                                                module=['numpy', 'scipy'], **kwargs)
```

Bases: [Expression](#)

Skew normal distribution component.

Asymmetric peak shape based on a normal distribution.

For definition see https://en.wikipedia.org/wiki/Skew_normal_distribution

See also <http://azzalini.stat.unipd.it/SN/>

$$f(x) = 2A\phi(x)\Phi(x)$$

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{t(x)^2}{2}\right]$$

$$\Phi(x) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{\alpha t(x)}{\sqrt{2}}\right) \right]$$

$$t(x) = \frac{x - x_0}{\omega}$$

Variable	Parameter
x_0	x0
A	A
ω	scale
α	shape

Parameters**x0**

[float] Location of the peak position (not maximum, which is given by the *mode* property).

A

[float] Height parameter of the peak.

scale

[float] Width (sigma) parameter.

shape: float

Skewness (asymmetry) parameter. For shape=0, the normal distribution (Gaussian) is obtained. The distribution is right skewed (longer tail to the right) if shape>0 and is left skewed if shape<0.

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Notes

The properties *mean* (position), *variance*, *skewness* and *mode* (position of maximum) are defined for convenience.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal*, *x1*, *x2*, *only_current=False*)

Estimate the skew normal distribution by calculating the momenta.

Parameters

signal

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns

bool

Notes

Adapted from Lin, Lee and Yen, *Statistica Sinica* 17, 909-927 (2007) <https://www.jstor.org/stable/24307705>

Examples

```
>>> g = hs.model.components1D.SkewNormal()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager._axes[-1].offset = -10
>>> s.axes_manager._axes[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True
```

property mean

Mean (position) of the component.

property mode

Mode (position of maximum) of the component.

property skewness

Skewness of the component.

property variance

Variance of the component.

class hyperspy.api.model.components1D.**SplitVoigt**(*A=1.0, sigma1=1.0, sigma2=1.0, fraction=0.0, centre=0.0*)

Bases: [Component](#)

Split pseudo-Voigt component.

$$pV(x, centre, \sigma) = (1 - \eta)G(x, centre, \sigma) + \eta L(x, centre, \sigma)$$

$$f(x) = \begin{cases} pV(x, centre, \sigma_1), & x \leq centre \\ pV(x, centre, \sigma_2), & x > centre \end{cases}$$

Variable	Parameter
<i>A</i>	A
η	fraction
σ_1	sigma1
σ_2	sigma2
<i>centre</i>	centre

Notes

This is a voigt function in which the upstream and downstream variance or sigma is allowed to vary to create an asymmetric profile. In this case the voigt is a pseudo voigt consisting of a mixed gaussian and lorentzian sum.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal*, *x1*, *x2*, *only_current=False*)

Estimate the split voigt function by calculating the momenta the gaussian.

Parameters**signal**

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns

bool

Notes

Adapted from <https://scipy-cookbook.readthedocs.io/items/FittingData.html>

Examples

```
>>> g = hs.model.components1D.SplitVoigt()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True
```

function(*x*)

Split pseudo voigt - a linear combination of gaussian and lorentzian

Parameters**x**

[array] independent variable

A

[float] area of pvoigt peak

center

[float] center position

sigma1

[float] standard deviation <= center position

sigma2

[float] standard deviation > center position

fraction

[float] weight for lorentzian peak in the linear combination, and (1-fraction) is the weight for gaussian peak.

function_nd(axis)

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly to obtain the fitted component without iterating over the navigation axes.

```
class hyperspy.api.model.components1D.Voigt(centre=10.0, area=1.0, gamma=0.2, sigma=0.1,
                                           module=['numpy', 'scipy'], **kwargs)
```

Bases: [Expression](#)

Voigt component.

Symmetric peak shape based on the convolution of a Lorentzian and Normal (Gaussian) distribution:

$$f(x) = G(x) \cdot L(x)$$

where $G(x)$ is the Gaussian function and $L(x)$ is the Lorentzian function. In this case using an approximate formula by David (see Notes). This approximation improves on the pseudo-Voigt function (linear combination instead of convolution of the distributions) and is, to a very good approximation, equivalent to a Voigt function:

$$z(x) = \frac{x + i\gamma}{\sqrt{2}\sigma}$$

$$w(z) = \frac{e^{-z^2} \operatorname{erfc}(-iz)}{\sqrt{2\pi}\sigma}$$

$$f(x) = A \cdot \Re \{w[z(x - x_0)]\}$$

Variable	Parameter
x_0	centre
A	area
γ	gamma
σ	sigma

Parameters**centre**

[float] Location of the maximum of the peak.

area

[float] Intensity below the peak.

gamma

[float] γ = HWHM of the Lorentzian distribution.

sigma: float

$2\sigma\sqrt{(2\log(2))}$ = FWHM of the Gaussian distribution.

****kwargs**

Extra keyword arguments are passed to the [Expression](#) component.

Notes

For convenience the *gwidth* and *lwidth* attributes can also be used to set and get the FWHM of the Gaussian and Lorentzian parts of the distribution, respectively. For backwards compatability, *FWHM* is another alias for the Gaussian width.

W.I.F. David, J. Appl. Cryst. (1986). 19, 63-64, doi:10.1107/S0021889886089999

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

estimate_parameters(*signal*, *x1*, *x2*, *only_current=False*)

Estimate the Voigt function by calculating the momenta of the Gaussian.

Parameters

signal

[Signal1D]

x1

[float] Defines the left limit of the spectral range to use for the estimation.

x2

[float] Defines the right limit of the spectral range to use for the estimation.

only_current

[bool] If False estimates the parameters for the full dataset.

Returns

bool

Exit status required for the *remove_background()* function.

Notes

Adapted from <https://scipy-cookbook.readthedocs.io/items/FittingData.html>

Examples

```
>>> g = hs.model.components1D.Voigt()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
True
```

23.4.2 components2D

Components that can be used to define a 2D model for e.g. curve fitting.

Writing a new template is easy, see the user guide documentation on creating components.

For more details see each component docstring.

Expression

Create a component from a string expression.

Gaussian2D

Normalized 2D elliptical Gaussian function component.

```
class hyperspy.api.model.components2D.Gaussian2D(A=1.0, sigma_x=1.0, sigma_y=1.0, centre_x=0.0,
        centre_y=0, module=None, **kwargs)
```

Bases: *Expression*

Normalized 2D elliptical Gaussian function component.

$$f(x, y) = \frac{A}{2\pi s_x s_y} \exp \left[-\frac{(x - x_0)^2}{2s_x^2} - \frac{(y - y_0)^2}{2s_y^2} \right]$$

Variable	Parameter
A	A
s_x, s_y	sigma_x/y
x_0, y_0	centre_x/y

Parameters

A

[float] Volume (height of the peak scaled by $2\pi s_x s_y$) – equivalent to the area in a 1D Gaussian.

sigma_x

[float] Width (scale parameter) of the Gaussian distribution in x direction.

sigma_y

[float] Width (scale parameter) of the Gaussian distribution in y direction.

centre_x

[float] Location of the Gaussian maximum (peak position) in x direction.

centre_y

[float] Location of the Gaussian maximum (peak position) in y direction.

add_rotation

[bool] If True, add the parameter *rotation_angle* corresponding to the angle between the x and the horizontal axis.

****kwargs**

Extra keyword arguments are passed to the *Expression* component.

Attributes

fwhm_x, fwhm_y

[float] Convenience attributes to get and set the full width at half maximum along the two axes.

Parameters**parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

23.5 hyperspy.api.plot

<code>hyperspy.api.plot.markers</code>	Markers that can be added to <i>BaseSignal</i> plots.
<code>hyperspy.api.plot.plot_histograms(signal_list)</code>	Plot the histogram of every signal in the list in one figure.
<code>hyperspy.api.plot.plot_images(images[, ...])</code>	Plot multiple images either as sub-images or overlaid in one figure.
<code>hyperspy.api.plot.plot_roi_map(signal[, rois])</code>	Plot one or multiple ROI maps of a signal.
<code>hyperspy.api.plot.plot_signals(signal_list)</code>	Plot several signals at the same time.
<code>hyperspy.api.plot.plot_spectra(spectra[, ...])</code>	Plot several spectra in the same figure.

23.5.1 hyperspy.api.plot.markers

<code>hyperspy.api.plot.markers.Arrows(offsets, U, V)</code>	A set of Arrow markers based on the <code>matplotlib.quiver.Quiver</code> class.
<code>hyperspy.api.plot.markers.Circles(offsets, sizes)</code>	A set of Circle Markers.
<code>hyperspy.api.plot.markers.Ellipses(offsets, ...)</code>	A set of Ellipse Markers
<code>hyperspy.api.plot.markers.HorizontalLines(...)</code>	A set of HorizontalLines markers
<code>hyperspy.api.plot.markers.Lines(segments[, ...])</code>	A set of Line Segments Markers.
<code>hyperspy.api.plot.markers.Markers(collection)</code>	A set of markers using Matplotlib collections.
<code>hyperspy.api.plot.markers.Points(offsets[, ...])</code>	A set of Points Markers.
<code>hyperspy.api.plot.markers.Polygons(verts[, ...])</code>	A Collection of Rectangles Markers
<code>hyperspy.api.plot.markers.Rectangles(...[, ...])</code>	A Collection of Rectangles Markers
<code>hyperspy.api.plot.markers.Squares(offsets, ...)</code>	A Collection of square markers.
<code>hyperspy.api.plot.markers.Texts(offsets[, ...])</code>	A set of text markers
<code>hyperspy.api.plot.markers.VerticalLines(...)</code>	A set of Vertical Line Markers

Markers that can be added to *BaseSignal* plots.

Examples

```
>>> import skimage
>>> im = hs.signals.Signal2D(skimage.data.camera())
>>> m = hs.plot.markers.Rectangles(
...     offsets=[10, 15],
...     widths=(5,),
...     heights=(7,),
...     angles=(0,),
...     color="red",
...     )
>>> im.add_marker(m)
```

```
class hyperspy.api.plot.markers.Arrows(offsets, U, V, C=None, scale=1, angles='xy', scale_units='xy',
                                       **kwargs)
```

Bases: [Markers](#)

A set of Arrow markers based on the `matplotlib.quiver.Quiver` class.

Initialize the set of Arrows Markers.

Parameters

offsets

[array_like] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

U

[array_like] The change in x (horizontal) direction for the arrows.

V

[array_like] The change in y (vertical) direction for the arrows.

C

[array_like or None]

kwargs

[dict] Keyword arguments are passed to `matplotlib.quiver.Quiver`.

update()

Update the markers on the plot.

```
class hyperspy.api.plot.markers.Circles(offsets, sizes, offset_transform='data', units='x',
                                       facecolors='none', **kwargs)
```

Bases: [Markers](#)

A set of Circle Markers.

Create a set of Circle Markers.

Parameters

offsets

[array_like] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

sizes

[numpy.ndarray] The size of the circles in units defined by the argument units.

facecolors

[matplotlib color or list of color] Set the facecolor(s) of the markers. It can be a color (all

patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence. If `c` is 'none', the patch will not be filled.

units

[{"points", "inches", "dots", "width", "height", "x", "y", "xy"}] The units in which majors and minors are given; "width" and "height" refer to the dimensions of the axes, while "x" and "y" refer to the *offsets* data units. "xy" differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `matplotlib.patches.Ellipse` with `axes.transData` as its transform.

kwargs

[dict] Keyword arguments are passed to `matplotlib.collections.CircleCollection`.

```
class hyperspy.api.plot.markers.Ellipses(offsets, heights, widths, angles=0, offset_transform='data',
                                         units='xy', **kwargs)
```

Bases: *Markers*

A set of Ellipse Markers

Initialize the set of Ellipse Markers.

Parameters**offsets**

[array_like] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

heights: array-like

The lengths of the second axes.

widths: array-like

The lengths of the first axes (e.g., major axis lengths).

angles

[array_like]

The angles of the first axes, degrees CCW from the x-axis.**units**

[{"points", "inches", "dots", "width", "height", "x", "y", "xy"}] The units in which majors and minors are given; "width" and "height" refer to the dimensions of the axes, while "x" and "y" refer to the *offsets* data units. "xy" differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `matplotlib.patches.Ellipse` with `axes.transData` as its transform.

kwargs:

Additional keyword arguments are passed to `matplotlib.collections.EllipseCollection`.

```
class hyperspy.api.plot.markers.HorizontalLines(offsets, **kwargs)
```

Bases: *Markers*

A set of HorizontalLines markers

Initialize a set of HorizontalLines markers.

Parameters**offsets**

[array_like] Positions of the markers

kwargs

[dict] Keyword arguments passed to the underlying marker collection. Any argument that is array-like and has *dtype=object* is assumed to be an iterating argument and is treated as such.

Examples

```
>>> import hyperspy.api as hs
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

```
>>> # Create a Signal2D with 2 navigation dimensions
>>> rng = np.random.default_rng(0)
>>> data = rng.random((25, 25, 100)) * 100
>>> s = hs.signals.Signal1D(data)
>>> offsets = np.array([10, 20, 40])
```

```
>>> m = hs.plot.markers.HorizontalLines(
...     offsets=offsets,
...     linewidth=4,
...     colors=['r', 'g', 'b'],
...     )
```

```
>>> s.plot()
>>> s.add_marker(m)
```

get_current_kwargs (*only_variable_length=False*)

Return the current keyword arguments for updating the collection.

Parameters**only_variable_length**

[bool] If True, only returns the variable length kwargs. Default is False.

Returns**kwargs**

[dict] The argument at the current navigation position.

class hyperspy.api.plot.markers.Lines(*segments, transform='data', **kwargs*)

Bases: *Markers*

A set of Line Segments Markers.

Initialize the set of Segments Markers.

Parameters**segments**

[numpy.ndarray] Must be with shape [n, 2, 2] ragged array with shape (n, 2, 3) at every navigation position. Defines the lines[[[x1,y1],[x2,y2]], ...] of the center of the ellipse.

kwargs

[dict] Additional keyword arguments are passed to `matplotlib.collections.LineCollection`.

Notes

Unlike markers using `offsets` argument, the positions of the segments are defined by the `segments` argument and the tranform specifying the coordinate system of the `segments` is `transform`.

```
class hyperspy.api.plot.markers.Markers(collection, offset_transform='data', transform='display',
                                       shift=None, plot_on_signal=True, name="",
                                       ScalarMappable_array=None, **kwargs)
```

Bases: `object`

A set of markers using Matplotlib collections.

The markers are defined by a set of arugment required by the collections, typically, `offsets`, `verts` or `segments` will define their positions.

To define a non-static marker any argument that can be set with the `matplotlib.collections.Collection.set()` method can be passed as an array with `dtype=object` of the constructor and the same size as the navigation axes of the a signal the markers will be added to.

Parameters

collection

[`matplotlib.collections.Collection` or `str`] A Matplotlib collection to be initialized.

offset_transform, transform

[`str`] `offset_transform` define the transformation used for the `offsets`` value and `transform` define the transformation for other arguments, typically to scale the size of the Path. It can be one of the following:

- "data": the offsets are defined in data coordinates and the `ax.transData` transformation is used.
- "relative": The offsets are defined in data coordinates in x and coordinates in y relative to the data plotted. Only for 1D figure.
- "axes": the offsets are defined in axes coordinates and the `ax.transAxes` transformation is used. (0, 0) is bottom left of the axes, and (1, 1) is top right of the axes.
- "xaxis": The offsets are defined in data coordinates in x and axes coordinates in y direction; use `matplotlib.axes.Axes.get_xaxis_transform()` transformation.
- "yaxis": The offsets are defined in data coordinates in y and axes coordinates in x direction; use `matplotlib.axes.Axes.get_xaxis_transform()` transformation..
- "display": the offsets are not transformed, i.e. are defined in the display coordinate system. (0, 0) is the bottom left of the window, and (width, height) is top right of the output in "display units" `matplotlib.transforms.IdentityTransform`.

shift

[`None` or `float`] Only for `offset_transform="relative"`. This applied a systematic shift in the y component of the `offsets` values. The shift is defined in the matplotlib "axes" coordinate system. This provides a constant shift from the data for labeling *Signal1D*.

plot_on_signal

[`bool`] If True, plot on signal figure, otherwise on navigator.

name

[`str`] The name of the markers.

ScalarMappable_array

[Array-like] Set the array of the `matplotlib.cm.ScalarMappable` of the matplotlib collection. The `ScalarMappable` array will overwrite `facecolor` and `edgecolor`. Default is `None`.

****kwargs**

[dict] Keyword arguments passed to the underlying marker collection. Any argument that is array-like and has `dtype=object` is assumed to be an iterating argument and is treated as such.

Examples

Add markers using a `matplotlib.collections.PatchCollection` which will display the specified subclass of `matplotlib.patches.Patch` at the position defined by the argument `offsets`.

```
>>> from matplotlib.collections import PatchCollection
>>> from matplotlib.patches import Circle
```

```
>>> m = hs.plot.markers.Markers(
...     collection=PatchCollection,
...     patches=[Circle((0, 0), 1)],
...     offsets=np.random.rand(10,2)*10,
...     )
>>> s = hs.signals.Signal2D(np.ones((10, 10, 10, 10)))
>>> s.plot()
>>> s.add_marker(m)
```

Adding star “iterating” markers using `matplotlib.collections.StarPolygonCollection`

```
>>> from matplotlib.collections import StarPolygonCollection
>>> import numpy as np
>>> rng = np.random.default_rng(0)
>>> data = np.ones((25, 25, 100, 100))
>>> s = hs.signals.Signal2D(data)
>>> offsets = np.empty(s.axes_manager.navigation_shape, dtype=object)
>>> for ind in np.ndindex(offsets.shape):
...     offsets[ind] = rng.random((10, 2)) * 100
```

Every other star has a size of 50/100

```
>>> m = hs.plot.markers.Markers(
...     collection=StarPolygonCollection,
...     offsets=offsets,
...     numsides=5,
...     color="orange",
...     sizes=(50, 100),
...     )
>>> s.plot()
>>> s.add_marker(m)
```

add_items(*navigation_indices=None, **kwargs*)

Add items to the markers.

Parameters

navigation_indices

[[tuple](#) or [None](#)] Only for variable-length markers. If [None](#), all for all navigation coordinates.

****kwargs**

[[dict](#)] Mapping of keys:values to add to the markers

Examples

Add a single item:

```
>>> offsets = np.array([[1, 1], [2, 2]])
>>> texts = np.array(["a", "b"])
>>> m = hs.plot.markers.Texts(offsets=offsets, texts=texts)
>>> print(m)
<Texts, length: 2>
>>> m.add_items(offsets=np.array([[0, 1]]), texts=["c"])
>>> print(m)
<Texts, length: 3>
```

Add a single item at a defined navigation position of variable length markers:

```
>>> offsets = np.empty(4, dtype=object)
>>> texts = np.empty(4, dtype=object)
>>> for i in range(len(offsets)):
...     offsets[i] = np.array([[1, 1], [2, 2]])
...     texts[i] = ['a' * (i+1)] * 2
>>> m = hs.plot.markers.Texts(offsets=offsets, texts=texts)
>>> m.add_items(
...     offsets=np.array([[0, 1]]), texts=["new_text"],
...     navigation_indices=(1, )
... )
```

close(*render_figure=True*)

Remove and disconnect the marker.

Parameters**render_figure**

[[bool](#), optional, default [True](#)] If [True](#), the figure is rendered after removing the marker. If [False](#), the figure is not rendered after removing the marker. This is useful when many markers are removed from a figure, since rendering the figure after removing each marker will slow things down.

classmethod from_signal(*signal, key=None, signal_axes='metadata', **kwargs*)

Initialize a marker collection from a hyperspy Signal.

Parameters

signal: `:class:`~.api.signals.BaseSignal``

A value passed to the Collection as {key:signal.data}

key: `str` or `None`

The key used to create a key value pair to create the subclass of `matplotlib.collections.Collection`. If `None` (default) the key is set to "offsets".

signal_axes: str, tuple of :class:`~.axes.UniformDataAxis` or None

If "metadata" look for signal_axes saved in metadata under s.metadata.Peaks. signal_axes and convert from pixel positions to real units before creating the collection. If a tuple of signal axes, those axes will be used otherwise (None) no transformation will happen.

get_current_kwargs(*only_variable_length=False*)

Return the current keyword arguments for updating the collection.

Parameters

only_variable_length

[bool] If True, only returns the variable length kwargs. Default is False.

Returns

kwargs

[dict] The argument at the current navigation position.

property offset_transform

The transform being used for the offsets values.

plot(*render_figure=True*)

Plot a marker which has been added to a signal.

Parameters

render_figure

[bool, optional, default True] If True, will render the figure after adding the marker. If False, the marker will be added to the plot, but will the figure will not be rendered. This is useful when plotting many markers, since rendering the figure after adding each marker will slow things down.

plot_colorbar()

Add a colorbar for the collection.

Returns

matplotlib.colorbar.Colorbar

The colorbar of the collection.

See also:

[*set_ScalarMappable_array*](#)

Examples

```
>>> rng = np.random.default_rng(0)
>>> s = hs.signals.Signal2D(np.ones((100, 100)))
>>> # Define the size of the circles
>>> sizes = rng.random((10, )) * 10 + 20
>>> # Define the position of the circles
>>> offsets = rng.random((10, 2)) * 100
>>> m = hs.plot.markers.Circles(
...     sizes=sizes,
...     offsets=offsets,
...     linewidth=2,
...     )
```

(continues on next page)

(continued from previous page)

```

>>> s.plot()
>>> s.add_marker(m)
>>> m.set_ScalarMappable_array(sizes.ravel() / 2)
>>> cbar = m.plot_colorbar()
>>> cbar.set_label('Circle radius')

```

remove_items(*indices*, *keys=None*, *navigation_indices=None*)

Remove items from the markers.

Parameters

indices

[*slice*, *int* or *numpy.ndarray*] Indicate indices of sub-arrays to remove along the specified axis.

keys

[*str*, *list* of *str* or *None*] Specify the key of the `Markers.kwargs` to remove. If *None*, use all compatible keys. Default is *None*.

navigation_indices

[*tuple*] Only for variable-length markers. If *None*, remove for all navigation coordinates.

Examples

Remove a single item:

```

>>> offsets = np.array([[1, 1], [2, 2]])
>>> m = hs.plot.markers.Points(offsets=offsets)
>>> print(m)
<Points, length: 2>
>>> m.remove_items(indices=(1, ))
>>> print(len(m))
1

```

Remove a single item at specific navigation position for variable length markers:

```

>>> offsets = np.empty(4, dtype=object)
>>> texts = np.empty(4, dtype=object)
>>> for i in range(len(offsets)):
...     offsets[i] = np.array([[1, 1], [2, 2]])
...     texts[i] = ['a' * (i+1)] * 2
>>> m = hs.plot.markers.Texts(offsets=offsets, texts=texts)
>>> m.remove_items(1, navigation_indices=(1, ))

```

Remove several items:

```

>>> offsets = np.stack([np.arange(0, 100, 10)]*2).T + np.array([5,]*2)
>>> texts = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'f', 'h', 'i'])
>>> m = hs.plot.markers.Texts(offsets=offsets, texts=texts)
>>> print(m)
<Texts, length: 10>
>>> m.remove_items(indices=[1, 2])
>>> print(m)
<Texts, length: 8>

```

set_ScalarMappable_array(*array*)

Set the array of the `matplotlib.cm.ScalarMappable` of the `matplotlib` collection. The `ScalarMappable` array will overwrite `facecolor` and `edgecolor`.

Parameters

array

[*array_like*] The value that are mapped to the colors.

See also:

[*plot_colorbar*](#)

property transform

The transform being used for the values other than offsets, typically sizes, etc.

update()

Update the markers on the plot.

class `hyperspy.api.plot.markers.Points`(*offsets*, *sizes=10*, *offset_transform='data'*, *units='points'*, ***kwargs*)

Bases: [*Markers*](#)

A set of Points Markers.

Initialize the set of points Markers.

Parameters

offsets

[*array_like*] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

sizes

[*int*, *float* or *array_like*, optional] The size of the markers in display coordinate system.

units

[{"points", "inches", "dots", "width", "height", "x", "y", "xy"}] The units in which majors and minors are given; "width" and "height" refer to the dimensions of the axes, while "x" and "y" refer to the *offsets* data units. "xy" differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `matplotlib.patches.Ellipse` with `axes.transData` as its transform.

kwargs

[*dict*] Keyword arguments are passed to `matplotlib.collections.CircleCollection`

class `hyperspy.api.plot.markers.Polygons`(*verts*, *transform='data'*, ***kwargs*)

Bases: [*Markers*](#)

A Collection of Rectangles Markers

Initialize the set of Segments Markers.

Parameters

verts

[*list* of `numpy.ndarray` or *list* of *list*] The verts define the vertices of the polygons. Note that this can be a ragged list and as such it is not automatically cast to a numpy array as

that would result in an array of objects. In the form `[[[x1,y1], [x2,y2], ... [xn, yn]],[x1,y1], [x2,y2], ... [xm,ym]], ...]`.

****kwargs**

`[dict]` Additional keyword arguments are passed to `matplotlib.collections.PolyCollection`

Notes

Unlike markers using `offsets` argument, the positions of the polygon are defined by the `verts` argument and the transform specifying the coordinate system of the `verts` is `transform`.

Examples

```
>>> import hyperspy.api as hs
>>> import numpy as np
>>> # Create a Signal2D with 2 navigation dimensions
>>> data = np.ones((25, 25, 100, 100))
>>> s = hs.signals.Signal2D(data)
>>> poylgon1 = [[1, 1], [20, 20], [1, 20], [25, 5]]
>>> poylgon2 = [[50, 60], [90, 40], [60, 40], [23, 60]]
>>> verts = [poylgon1, poylgon2]
>>> # Create the markers
>>> m = hs.plot.markers.Polygons(
...     verts=verts,
...     linewidth=3,
...     facecolors=('g',),
...     )
>>> # Add the marker to the signal
>>> s.plot()
>>> s.add_marker(m)
```

class `hyperspy.api.plot.markers.Rectangles`(*offsets, widths, heights, angles=0, offset_transform='data', units='xy', **kwargs*)

Bases: [Markers](#)

A Collection of Rectangles Markers

Initialize the set of Segments Markers.

Parameters

offsets

`[array_like]` The positions `[x, y]` of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

heights: array-like

The lengths of the second axes.

widths: array-like

The lengths of the first axes (e.g., major axis lengths).

angles

`[array_like]`

The angles of the first axes, degrees CCW from the x-axis.

units

[{"points", "inches", "dots", "width'", "height", "x", "y", "xy"}] The units in which majors and minors are given; "width" and "height" refer to the dimensions of the axes, while "x" and "y" refer to the *offsets* data units. "xy" differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `matplotlib.patches.Ellipse` with `axes.transData` as its transform.

kwargs:

Additional keyword arguments are passed to `hyperspy.external.matplotlib.collections.RectangleCollection`.

```
class hyperspy.api.plot.markers.Squares(offsets, widths, angles=0, offset_transform='data', units='x',
                                       **kwargs)
```

Bases: *Markers*

A Collection of square markers.

Initialize the set of square Markers.

Parameters

offsets

[array_like] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

widths: array-like

The lengths of the first axes (e.g., major axis lengths).

angles

[array_like]

The angles of the first axes, degrees CCW from the x-axis.

units

[{"points", "inches", "dots", "width'", "height", "x", "y", "xy"}] The units in which majors and minors are given; "width" and "height" refer to the dimensions of the axes, while "x" and "y" refer to the *offsets* data units. "xy" differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `matplotlib.patches.Ellipse` with `axes.transData` as its transform.

kwargs:

Additional keyword arguments are passed to `hyperspy.external.matplotlib.collections.SquareCollection`.

```
class hyperspy.api.plot.markers.Texts(offsets, offset_transform='data', transform='display', **kwargs)
```

Bases: *Markers*

A set of text markers

Initialize the set of Circle Markers.

Parameters

offsets

[array_like] The positions [x, y] of the center of the marker. If the offsets are not provided, the marker will be placed at the current navigation position.

sizes

[array_like] The size of the text in points.

facecolors

[(list of) matplotlib color] Set the facecolor(s) of the markers. It can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence. If *c* is 'none', the patch will not be filled.

kwargs

[dict] Keyword arguments are passed to `matplotlib.collections.CircleCollection`.

class `hyperspy.api.plot.markers.VerticalLines`(*offsets*, ***kwargs*)

Bases: *Markers*

A set of Vertical Line Markers

Initialize the set of Vertical Line Markers.

Parameters

x: [n]

Positions of the markers

kwargs: dict

Keyword arguments passed to the underlying marker collection. Any argument that is array-like and has *dtype=object* is assumed to be an iterating argument and is treated as such.

Examples

```
>>> import hyperspy.api as hs
>>> import numpy as np
>>> # Create a Signal2D with 2 navigation dimensions
>>> rng = np.random.default_rng(0)
>>> data = rng.random((25, 25, 100))
>>> s = hs.signals.Signal1D(data)
>>> offsets = np.array([10, 20, 40])
>>> # Create the markers
>>> m = hs.plot.markers.VerticalLines(
...     offsets=offsets,
...     linewidth=3,
...     colors=['r', 'g', 'b'],
... )
>>> # Add the marker to the signal
>>> s.plot()
>>> s.add_marker(m)
```

get_current_kwargs(*only_variable_length=False*)

Return the current keyword arguments for updating the collection.

Parameters

only_variable_length

[bool] If True, only returns the variable length kwargs. Default is False.

Returns

kwargs

[dict] The argument at the current navigation position.

Plotting functions.

Functions:

plot_images

Plot multiple images in the same figure.

plot_spectra

Plot multiple spectra in the same figure.

plot_signals

Plot multiple signals at the same time.

plot_histograms

Compute and plot the histograms of multiple signals in the same figure.

plot_roi_map

Plot a the spatial variation of a signal sliced by a ROI in the signal space.

The `hyperspy.api.plot` module contains the following submodules:

`hyperspy.api.plot.markers`

Markers that can be added to `BaseSignal` figure.

`hyperspy.api.plot.plot_histograms(signal_list, bins='fd', range_bins=None, color=None, linestyle=None, legend='auto', fig=None, **kwargs)`

Plot the histogram of every signal in the list in one figure.

This function creates a histogram for each signal and plots the list with the `plot_spectra()` function.

Parameters

signal_list

[iterable] Ordered list of spectra to plot. If `style` is "cascade" or "mosaic", the spectra can have different size and axes.

bins

[int, list or str, optional] If `bins` is a string, then it must be one of:

- 'knuth' : use Knuth's rule to determine bins,
- 'scott' : use Scott's rule to determine bins,
- 'fd' : use the Freedman-diaconis rule to determine bins,
- 'blocks' : use bayesian blocks for dynamic bin widths.

range_bins

[None or tuple, optional] The minimum and maximum range for the histogram. If not specified, it will be `(x.min(), x.max())`.

color

[None, (list of) matplotlib color, optional] Sets the color of the lines of the plots. For a list, if its length is less than the number of spectra to plot, the colors will be cycled. If `None`, use default matplotlib color cycle.

linestyle

[None, (list of) matplotlib line style, optional] The main line styles are '-', '--', '-.', ':'. For a list, if its length is less than the number of spectra to plot, linestyle will be cycled. If `None`, use continuous lines (same as '-').

legend

[None, list of str, 'auto', default 'auto'] Display a legend. If 'auto', the title of each spectra (`metadata.General.title`) is used.

legend_picking

[`bool`, default `True`] If `True`, a spectrum can be toggled on and off by clicking on the line in the legend.

fig

[`None`, `matplotlib.figure.Figure`, default `None`] If `None` (default), a default figure will be created.

****kwargs**

other keyword arguments (weight and density) are described in `numpy.histogram()`.

Returns

`matplotlib.axes.Axes` or list of `matplotlib.axes.Axes`

An array is returned when `style='mosaic'`.

Examples

Histograms of two random chi-square distributions.

```
>>> img = hs.signals.Signal2D(np.random.chisquare(1, [10, 10, 100]))
>>> img2 = hs.signals.Signal2D(np.random.chisquare(2, [10, 10, 100]))
>>> hs.plot.plot_histograms([img, img2], legend=['hist1', 'hist2'])
<Axes: xlabel='value (<undefined>)', ylabel='Intensity'>
```

```
hyperspy.api.plot.plot_images(images, cmap=None, no_nans=False, per_row=3, label='auto',
                              labelwrap=30, suptitle=None, suptitle_fontsize=18, colorbar='default',
                              centre_colormap='auto', scalebar=None, scalebar_color='white',
                              axes_decor='all', padding=None, tight_layout=False, aspect='auto',
                              min_asp=0.1, namefrac_thresh=0.4, fig=None, vmin=None, vmax=None,
                              overlay=False, colors='auto', alphas=1.0, legend_picking=True,
                              legend_loc='upper right', pixel_size_factor=None, **kwargs)
```

Plot multiple images either as sub-images or overlaid in one figure.

Parameters**images**

[list of `Signal2D` or `BaseSignal`] *images* should be a list of Signals to plot. For `BaseSignal` with navigation dimensions 2 and signal dimension 0, the signal will be transposed to form a `Signal2D`. Multi-dimensional images will have each plane plotted as a separate image. If any of the signal shapes is not suitable, a `ValueError` will be raised.

cmap

[`None`, (list of) `matplotlib.colors.Colormap` or `str`, default `None`] The colormap used for the images, by default uses the setting color map `signal` from the plot preferences. A list of colormaps can also be provided, and the images will cycle through them. Optionally, the value `'mpl_colors'` will cause the `cmap` to loop through the default `matplotlib` colors (to match with the default output of the `plot_spectra()` method). Note: if using more than one colormap, using the `'single'` option for `colorbar` is disallowed.

no_nans

[`bool`, optional] If `True`, set nans to zero for plotting.

per_row

[`int`, optional] The number of plots in each row.

label

[`None`, `str`, `list` of `str`, optional] Control the title labeling of the plotted images. If `None`, no titles will be shown. If `'auto'` (default), function will try to determine suitable titles using Signal2D titles, falling back to the `'titles'` option if no good short titles are detected. Works best if all images to be plotted have the same beginning to their titles. If `'titles'`, the title from each image's `metadata.General.title` will be used. If any other single `str`, images will be labeled in sequence using that `str` as a prefix. If a list of `str`, the list elements will be used to determine the labels (repeated, if necessary).

labelwrap

[`int`, optional] Integer specifying the number of characters that will be used on one line. If the function returns an unexpected blank figure, lower this value to reduce overlap of the labels between figures.

suptitle

[`str`, optional] Title to use at the top of the figure. If called with `label='auto'`, this parameter will override the automatically determined title.

suptitle_fontsize

[`int`, optional] Font size to use for super title at top of figure.

colorbar

[`'default'`, `'multi'`, `'single'`, `None`, optional] Controls the type of colorbars that are plotted, incompatible with `overlay=True`. If `'default'`, same as `'multi'` when `overlay=False`, otherwise same as `None`. If `'multi'`, individual colorbars are plotted for each (non-RGB) image. If `'single'`, all (non-RGB) images are plotted on the same scale, and one colorbar is shown for all. If `None`, no colorbar is plotted.

centre_colormap

[`'auto'`, `True`, `False`, optional] If `True`, the centre of the color scheme is set to zero. This is particularly useful when using diverging color schemes. If `'auto'` (default), diverging color schemes are automatically centred.

scalebar

[`None`, `'all'`, `list` of `int`, optional] If `None` (or `False`), no scalebars will be added to the images. If `'all'`, scalebars will be added to all images. If list of ints, scalebars will be added to each image specified.

scalebar_color

[`str`, optional] A valid MPL color string; will be used as the scalebar color.

axes_decor

[`'all'`, `'ticks'`, `'off'`, `None`, optional] Controls how the axes are displayed on each image; default is `'all'`. If `'all'`, both ticks and axis labels will be shown. If `'ticks'`, no axis labels will be shown, but ticks/labels will. If `'off'`, all decorations and frame will be disabled. If `None`, no axis decorations will be shown, but ticks/frame will.

padding

[`None`, `dict`, optional] This parameter controls the spacing between images. If `None`, default options will be used. Otherwise, supply a dictionary with the spacing options as keywords and desired values as values. Values should be supplied as used in `matplotlib.pyplot.subplots_adjust()`, and can be `'left'`, `'bottom'`, `'right'`, `'top'`, `'wspace'` (width) and `'hspace'` (height).

tight_layout

[`bool`, optional] If `true`, hyperspy will attempt to improve image placement in figure using `matplotlib's tight_layout`. If `false`, repositioning images inside the figure will be left as an exercise for the user.

aspect

[`str`, `float`, `int`, optional] If 'auto', aspect ratio is auto determined, subject to `min_asp`. If 'square', image will be forced onto square display. If 'equal', aspect ratio of 1 will be enforced. If float (or int/long), given value will be used.

min_asp

[`float`, optional] Minimum aspect ratio to be used when plotting images.

namefrac_thresh

[`float`, optional] Threshold to use for auto-labeling. This parameter controls how much of the titles must be the same for the auto-shortening of labels to activate. Can vary from 0 to 1. Smaller values encourage shortening of titles by auto-labeling, while larger values will require more overlap in titles before activating the auto-label code.

fig

[`matplotlib.figure.Figure`, default `None`] If set, the images will be plotted to an existing matplotlib figure.

vmin, vmax: scalar, str, None

If `str`, formatted as 'xth', use this value to calculate the percentage of pixels that are left out of the lower and upper bounds. For example, for a `vmin` of '1th', 1% of the lowest will be ignored to estimate the minimum value. Similarly, for a `vmax` value of '1th', 1% of the highest value will be ignored in the estimation of the maximum value. It must be in the range [0, 100]. See `numpy.percentile()` for more explanation. If `None`, use the percentiles value set in the preferences. If float or integer, keep this value as bounds. Note: `vmin` is ignored when overlaying images.

overlay

[`bool`, optional] If True, overlays the images with different colors rather than plotting each image as a subplot.

colors

['auto', `list` of `str`, optional] If list, it must contains colors acceptable to matplotlib [1]. If 'auto', colors will be taken from `matplotlib.colors.BASE_COLORS`.

alphas

[`float` or `list` of `float`, optional] Float value or a list of floats corresponding to the alpha value of each color.

legend_picking: bool, optional

If True (default), an image can be toggled on and off by clicking on the legended line. For `overlay=True` only.

legend_loc

[`str`, `int`, optional] This parameter controls where the legend is placed on the figure see the `matplotlib.pyplot.legend()` docstring for valid values

pixel_size_factor

[`None`, `int` or `float`, optional] If `None` (default), the size of the figure is taken from the `matplotlib.rcParams`. Otherwise sets the size of the figure when plotting an overlay image. The higher the number the larger the figure and therefore a greater number of pixels are used. This value will be ignored if a `Figure` is provided.

****kwargs, optional**

Additional keyword arguments passed to `matplotlib.pyplot.imshow()`.

Returns**axes_list**

[`list`] A list of subplot axes that hold the images.

See also:***plot_spectra***

Plotting of multiple spectra

plot_signals

Plotting of multiple signals

plot_histograms

Compare signal histograms

Notes

interpolation is a useful parameter to provide as a keyword argument to control how the space between pixels is interpolated. A value of 'nearest' will cause no interpolation between pixels.

tight_layout is known to be quite brittle, so an option is provided to disable it. Turn this option off if output is not as expected, or try adjusting *label*, *labelwrap*, or *per_row*.

References

[1]

`hyperspy.api.plot.plot_roi_map(signal, rois=1)`

Plot one or multiple ROI maps of a `signal`.

Uses regions of interest (ROIs) to select ranges along the signal axis.

For each ROI, a plot is generated of the summed data values within this signal ROI at each point in the `signal`'s navigator.

The ROIs can be moved interactively and the corresponding plots will update automatically.

Parameters

signal: `:class:`~.api.signals.BaseSignal``

The signal to inspect.

rois: `int`, list of `:class:`~.api.roi.SpanROI`` or `:class:`~.api.roi.RectangularROI``

ROIs that represent colour channels in map. Can either pass a list of ROI objects, or an `int`, `N`, in which case `N` ROIs will be created. Currently limited to a maximum of 3 ROIs.

Returns

all_sum

`[BaseSignal]` Sum over all positions (navigation dimensions) of the signal, corresponds to the 'navigator' (in signal space) on which the ROIs are added.

rois

`[list of SpanROI or RectangularROI]` The ROI objects that slice signal.

roi_signals

`[BaseSignal]` Slices of signal corresponding to each ROI.

roi_sums

`[BaseSignal]` The summed `roi_signals`.

Examples

3D hyperspectral data:

For 3D hyperspectral data, the ROIs used will be instances of *SpanROI*. Therefore, these ROIs can be used to select particular spectral ranges, e.g. a particular peak.

The map generated for a given ROI is therefore the sum of this spectral region at each point in the hyperspectral map. Therefore, regions of the sample where this peak is bright will be bright in this map.

4D STEM:

For 4D STEM data, the ROIs used will be instances of *RectangularROI*. These ROIs can be used to select particular regions in reciprocal space, e.g. a particular diffraction spot.

The map generated for a given ROI is the intensity of this region at each point in the scan. Therefore, regions of the scan where a particular spot is intense will appear bright.

```
hyperspy.api.plot.plot_signals(signal_list, sync=True, navigator='auto', navigator_list=None, **kwargs)
```

Plot several signals at the same time.

Parameters

signal_list

[list of *BaseSignal*] If sync is set to True, the signals must have the same navigation shape, but not necessarily the same signal shape.

sync

[bool, optional] If True (default), the signals will share navigation. All the signals must have the same navigation shape for this to work, but not necessarily the same signal shape.

navigator

[None, *BaseSignal* or str]

{'auto' | 'spectrum' | 'slider'}, default 'auto'

See signal.plot docstring for full description.

navigator_list

[None, list of *BaseSignal* or list of str, default None] Set different navigator options for the signals. Must use valid navigator arguments: 'auto', None, 'spectrum', 'slider', or a HyperSpy Signal. The list must have the same size as signal_list. If None, the argument specified in navigator will be used.

**kwargs

[dict] Any extra keyword arguments are passed to each signal plot method.

Examples

```
>>> s1 = hs.signals.Signal1D(np.arange(100).reshape((10, 10)))
>>> s2 = hs.signals.Signal1D(np.arange(100).reshape((10, 10)) * -1)
>>> hs.plot.plot_signals([s1, s2])
```

Specifying the navigator:

```
>>> hs.plot.plot_signals([s1, s2], navigator="slider")
```

Specifying the navigator for each signal:

```

>>> s3 = hs.signals.Signal1D(np.ones((10, 10)))
>>> s_nav = hs.signals.Signal1D(np.ones((10)))
>>> hs.plot.plot_signals(
...     [s1, s2, s3], navigator_list=["slider", None, s_nav]
...     )

```

`hyperspy.api.plot.plot_spectra`(*spectra*, *style*='overlap', *color*=None, *linestyle*=None, *drawstyle*='default', *padding*=1.0, *legend*=None, *legend_picking*=True, *legend_loc*='upper right', *fig*=None, *ax*=None, *auto_update*=None, ***kwargs*)

Plot several spectra in the same figure.

Parameters

spectra

[list of *Signal1D* or *BaseSignal*] Ordered spectra list of signal to plot. If *style* is “cascade” or “mosaic”, the spectra can have different size and axes. For *BaseSignal* with navigation dimensions 1 and signal dimension 0, the signal will be transposed to form a *Signal1D*.

style

[{'overlap' | 'cascade' | 'mosaic' | 'heatmap'}, default 'overlap'] The style of the plot: 'overlap' (default), 'cascade', 'mosaic', or 'heatmap'.

color

[None or (list of) matplotlib color, default None] Sets the color of the lines of the plots (no action on 'heatmap'). For a list, if its length is less than the number of spectra to plot, the colors will be cycled. If None (default), use default matplotlib color cycle.

linestyle

[None or (list of) matplotlib line style, default None] Sets the line style of the plots (no action on 'heatmap'). The main line style are '-', '--', '-.', ':'. For a list, if its length is less than the number of spectra to plot, linestyle will be cycled. If None, use continuous lines (same as '-').

drawstyle

[{'default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post'},] default 'default' The drawstyle determines how the points are connected, no action with *style*='heatmap'. See `matplotlib.lines.Line2D.set_drawstyle()` for more information. The 'default' value is defined by matplotlib.

padding

[float, default 1.0] Option for “cascade”. 1 guarantees that there is no overlapping. However, in many cases, a value between 0 and 1 can produce a tighter plot without overlapping. Negative values have the same effect but reverse the order of the spectra without reversing the order of the colors.

legend

[None, list of str, 'auto', default None] If list of string, legend for 'cascade' or title for 'mosaic' is displayed. If 'auto', the title of each spectra (`metadata.General.title`) is used. Default None.

legend_picking

[bool, default True] If True (default), a spectrum can be toggled on and off by clicking on the legended line.

legend_loc

[str or int, optional] This parameter controls where the legend is placed on the figure; see the `pyplot.legend` docstring for valid values. Default 'upper right'.

fig

[None, `matplotlib.figure.Figure`, default None] If None (default), a default figure will be created. Specifying `fig` will not work for the 'heatmap' style.

ax

[None, `matplotlib.axes.Axes`, default None] If None (default), a default ax will be created. Will not work for 'mosaic' or 'heatmap' style.

auto_update

[bool or None, default None] If True, the plot will update when the data are changed. Only supported with style='overlap' and a list of signal with navigation dimension 0. If None (default), update the plot only for style='overlap'.

****kwargs**

[dict] Depending on the style used, the keyword arguments are passed to different functions

- "overlap" or "cascade": arguments passed to `matplotlib.pyplot.figure()`
- "mosiac": arguments passed to `matplotlib.pyplot.subplots()`
- "heatmap": arguments passed to `plot()`.

Returns

`matplotlib.axes.Axes` or list of `matplotlib.axes.Axes`

An array is returned when `style` is 'mosaic'.

Examples

```
>>> s = hs.signals.Signal1D(np.arange(100).reshape((10, 10)))
>>> hs.plot.plot_spectra(s, style='cascade', color='red', padding=0.5)
<Axes: xlabel='<undefined>' (<undefined>)'>
```

To save the plot as a png-file

```
>>> ax = hs.plot.plot_spectra(s)
>>> ax.figure.savefig("test.png")
```

23.6 hyperspy.api.roi

<code>hyperspy.api.roi.CircleROI</code> ([cx, cy, r, r_inner])	Selects a circular or annular region in a 2D space.
<code>hyperspy.api.roi.Line2DROI</code> ([x1, y1, x2, y2, ...])	Selects a line of a given width in 2D space.
<code>hyperspy.api.roi.Point1DROI</code> ([value])	Selects a single point in a 1D space.
<code>hyperspy.api.roi.Point2DROI</code> ([x, y])	Selects a single point in a 2D space.
<code>hyperspy.api.roi.RectangularROI</code> ([left, top, ...])	Selects a range in a 2D space.
<code>hyperspy.api.roi.SpanROI</code> ([left, right])	Selects a range in a 1D space.

Region of interests (ROIs).

ROIs operate on *BaseSignal* instances and include widgets for interactive operation.

The following 1D ROIs are available:

Point1DROI

Single element ROI of a 1D signal.

SpanROI

Interval ROI of a 1D signal.

The following 2D ROIs are available:

Point2DROI

Single element ROI of a 2D signal.

RectangularROI

Rectagular ROI of a 2D signal.

CircleROI

(Hollow) circular ROI of a 2D signal

Line2DROI

Line profile of a 2D signal with customisable width.

class hyperspy.api.roi.CircleROI(*cx=None, cy=None, r=None, r_inner=0*)

Bases: *BaseInteractiveROI*

Selects a circular or annular region in a 2D space. The coordinates of the center of the circle are stored in the 'cx' and 'cy' attributes. The radius is in the *r* attribute. If an internal radius is defined using the *r_inner* attribute, then an annular region is selected instead. *CircleROI* can be used in place of a tuple containing (*cx, cy, r*), (*cx, cy, r, r_inner*) when *r_inner* is not *None*.

Sets up events.changed event, and inits HasTraits.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters**display**

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str, iterable of str or None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

is_valid()

Determine if the ROI is in a valid state.

This is typically determined by all the coordinates being defined, and that the values makes sense relative to each other.

class hyperspy.api.roi.Line2DROI(*x1=None, y1=None, x2=None, y2=None, linewidth=0*)

Bases: *BaseInteractiveROI*

Selects a line of a given width in 2D space. The coordinates of the end points of the line are stored in the *x1, y1, x2, y2* parameters. The length is available in the *length* parameter and the method *angle* computes the angle of the line with the axes.

Line2DROI can be used in place of a tuple containing the coordinates of the two end-points of the line and the linewidth (*x1, y1, x2, y2, linewidth*).

Sets up events.changed event, and inits HasTraits.

angle (*axis='horizontal', units='degrees'*)

“Angle between ROI line and selected axis

Parameters

axis

[*str*, {'horizontal', 'vertical'}, optional] Select axis against which the angle of the ROI line is measured. 'x' is alias to 'horizontal' and 'y' is 'vertical' (Default: 'horizontal')

units

[*str*, {'degrees', 'radians'}] The angle units of the output (Default: 'degrees')

Returns**angle**

[*float*]

Examples

```
>>> r = hs.roi.Line2DROI(0., 0., 1., 2.)
>>> r.angle()
63.43494882292201
```

gui (*display=True*, *toolkit=None*, ***kwargs*)

Display or return interactive GUI element if available.

Parameters**display**

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str*, iterable of *str* or *None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

static profile_line (*img*, *src*, *dst*, *axes*, *linewidth=1*, *order=1*, *mode='constant'*, *cval=0.0*)

Return the intensity profile of an image measured along a scan line.

Parameters**img**

[*numpy.ndarray*] The image, either grayscale (2D array) or multichannel (3D array, where the final axis contains the channel information).

src

[*tuple* of *float*, *tuple* of *int*] The start point of the scan line. Length of tuple is 2.

dst

[*tuple* of *float*, *tuple* of *int*] The end point of the scan line. Length of tuple is 2.

linewidth

[*int*, optional] Width of the scan, perpendicular to the line

order

[{0, 1, 2, 3, 4, 5}, optional] The order of the spline interpolation to compute image values at non-integer coordinates. 0 means nearest-neighbor interpolation.

mode

[{'constant', 'nearest', 'reflect', 'wrap'}, optional] How to compute any values falling outside of the image.

cval

[*float*, optional] If *mode='constant'*, what constant value to use outside the image.

Returns**numpy.ndarray**

The intensity profile along the scan line. The length of the profile is the ceil of the computed length of the scan line.

Notes

The destination point is included in the profile, in contrast to standard numpy indexing. Requires uniform navigation axes.

class hyperspy.api.roi.**Point1DROI**(*value=None*)

Bases: BasePointROI

Selects a single point in a 1D space. The coordinate of the point in the 1D space is stored in the 'value' trait.

Point1DROI can be used in place of a tuple containing the value of value.

Examples

```
>>> roi = hs.roi.Point1DROI(0.5)
>>> value, = roi
>>> print(value)
0.5
```

Sets up events.changed event, and inits HasTraits.

gui(*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters**display**

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

class hyperspy.api.roi.**Point2DROI**(*x=None, y=None*)

Bases: BasePointROI

Selects a single point in a 2D space. The coordinates of the point in the 2D space are stored in the traits 'x' and 'y'.

Point2DROI can be used in place of a tuple containing the coordinates of the point (x, y).

Examples

```
>>> roi = hs.roi.Point2DROI(3, 5)
>>> x, y = roi
>>> print(x, y)
3.0 5.0
```

Sets up events.changed event, and inits HasTraits.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str, iterable of str or None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

class hyperspy.api.roi.RectangularROI (*left=None, top=None, right=None, bottom=None*)

Bases: *BaseInteractiveROI*

Selects a range in a 2D space. The coordinates of the range in the 2D space are stored in the traits 'left', 'right', 'top' and 'bottom'. Convenience properties 'x', 'y', 'width' and 'height' are also available, but cannot be used for initialization.

RectangularROI can be used in place of a tuple containing (left, right, top, bottom).

Examples

```
>>> roi = hs.roi.RectangularROI(left=0, right=10, top=20, bottom=20.5)
>>> left, right, top, bottom = roi
>>> print(left, right, top, bottom)
0.0 10.0 20.0 20.5
```

Sets up events.changed event, and inits HasTraits.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str, iterable of str or None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

property height

Returns / sets the height of the ROI

is_valid()

Determine if the ROI is in a valid state.

This is typically determined by all the coordinates being defined, and that the values makes sense relative to each other.

property width

Returns / sets the width of the ROI

property x

Returns / sets the x coordinate of the ROI without changing its width

property y

Returns / sets the y coordinate of the ROI without changing its height

class `hyperspy.api.roi.SpanROI` (*left=None, right=None*)

Bases: *BaseInteractiveROI*

Selects a range in a 1D space. The coordinates of the range in the 1D space are stored in the traits 'left' and 'right'.

SpanROI can be used in place of a tuple containing the left and right values.

Examples

```
>>> roi = hs.roi.SpanROI(-3, 5)
>>> left, right = roi
>>> print(left, right)
-3.0 5.0
```

Sets up events.changed event, and inits HasTraits.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters**display**

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str, iterable of str or None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

is_valid()

Determine if the ROI is in a valid state.

This is typically determined by all the coordinates being defined, and that the values makes sense relative to each other.

23.7 hyperspy.api.samfire

`hyperspy.api.samfire.fit_tests`

`hyperspy.api.samfire.global_strategies`

`hyperspy.api.samfire.local_strategies`

`hyperspy.api.samfire.SamfirePool(**kwargs)` Creates and manages a pool of SAMFire workers.

23.7.1 hyperspy.api.samfire.fit_tests

`hyperspy.api.samfire.fit_tests.AIC_test(...)` Akaike information criterion test

`hyperspy.api.samfire.fit_tests.AICc_test(...)` Akaike information criterion (with a correction) test

`hyperspy.api.samfire.fit_tests.BIC_test(...)` Bayesian information criterion test

`hyperspy.api.samfire.fit_tests.red_chisq_test(...)`

class `hyperspy.api.samfire.fit_tests.AIC_test(tolerance)`

Bases: `object`

Akaike information criterion test

map(*model*, *mask*)

test(*model*, *ind*)

class `hyperspy.api.samfire.fit_tests.AICc_test(tolerance)`

Bases: `object`

Akaike information criterion (with a correction) test

map(*model*, *mask*)

test(*model*, *ind*)

class `hyperspy.api.samfire.fit_tests.BIC_test(tolerance)`

Bases: `object`

Bayesian information criterion test

map(*model*, *mask*)

test(*model*, *ind*)

class `hyperspy.api.samfire.fit_tests.red_chisq_test(tolerance)`

Bases: `goodness_test`

map(*model*, *mask*)

test(*model*, *ind*)

property tolerance

The tolerance.

23.7.2 hyperspy.api.samfire.global_strategies

<code>hyperspy.api.samfire.global_strategies.GlobalStrategy(name)</code>	A SAMFire strategy that operates in "parameter space" - i.e the pixel positions are not important, and only parameter value distributions are segmented to be used as starting point estimators.
<code>hyperspy.api.samfire.global_strategies.HistogramStrategy([bins])</code>	

class hyperspy.api.samfire.global_strategies.**GlobalStrategy**(*name*)

Bases: `SamfireStrategy`

A SAMFire strategy that operates in "parameter space" - i.e the pixel positions are not important, and only parameter value distributions are segmented to be used as starting point estimators.

clean()

Purges the currently saved values (not the database).

plot(*fig=None*)

Plots the current database of histograms

Parameters

fig

[None of `HistogramTilePlot`] If given updates the plot.

refresh(*overwrite, given_pixels=None*)

Refreshes the database (i.e. constructs it again from scratch)

remove()

Removes this strategy from its SAMFire

update(*ind, isgood*)

Updates the database and marker with the given pixel results

Parameters

ind

[tuple] the index with new results

isgood

[bool] if the fit was successful.

values(*ind=None*)

Returns the saved most frequent values that should be used for prediction

class hyperspy.api.samfire.global_strategies.**HistogramStrategy**(*bins='fd'*)

Bases: `GlobalStrategy`

clean()

Purges the currently saved values (not the database).

plot(*fig=None*)

Plots the current database of histograms

Parameters

fig

[None of `HistogramTilePlot`] If given updates the plot.

refresh(*overwrite*, *given_pixels=None*)

Refreshes the database (i.e. constructs it again from scratch)

remove()

Removes this strategy from its SAMFire

update(*ind*, *isgood*)

Updates the database and marker with the given pixel results

Parameters

ind

[[tuple](#)] the index with new results

isgood

[[bool](#)] if the fit was successful.

values(*ind=None*)

Returns the saved most frequent values that should be used for prediction

23.7.3 hyperspy.api.samfire.local_strategies

hyperspy.api.samfire.local_strategies.LocalStrategy (name)	A SAMFire strategy that operates in "pixel space" - i.e calculates the starting point estimates based on the local averages of the pixels.
hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy ()	Reduced chi-squared Local strategy of the SAMFire.

class `hyperspy.api.samfire.local_strategies.LocalStrategy`(*name*)

Bases: `SamfireStrategy`

A SAMFire strategy that operates in "pixel space" - i.e calculates the starting point estimates based on the local averages of the pixels. Requires some weighting method (e.g. reduced chi-squared).

clean()

Purges the currently saved values.

plot(*fig=None*)

Plots the current marker in a flat image

Parameters

fig

[[hyperspy.api.signals.Signal2D](#) or `numpy.ndarray`] if an already plotted image, then updates. Otherwise creates a new one.

Returns

[hyperspy.api.signals.Signal2D](#)

The resulting 2D signal. If passed again, will be updated (computationally cheaper operation).

property radii

A tuple of ≥ 0 floats that show the "radii of relevance"

refresh(*overwrite*, *given_pixels=None*)

Refreshes the marker - recalculates with the current values from scratch.

Parameters

overwrite

[bool] If True, all but the *given_pixels* will be recalculated. Used when part of already calculated results has to be refreshed. If False, only use pixels with marker == -scale (by default -1) to propagate to pixels with marker >= 0. This allows “ignoring” pixels with marker < -scale (e.g. -2).

given_pixels

[`numpy.ndarray` of bool] Pixels with True value are assumed as correctly calculated.

remove()

Removes this strategy from its SAMFire

property samf

The SAMFire that owns this strategy.

update(*ind*, *isgood*)

Updates the database and marker with the given pixel results

Parameters

ind

[tuple] the index with new results

isgood

[bool] if the fit was successful.

values(*ind*)

Returns the current starting value estimates for the given pixel. Calculated as the weighted local average. Only returns components that are active, and parameters that are free.

Parameters

ind

[tuple] the index of the pixel of interest.

Returns

values

[dict] A dictionary of estimates, structured as {component_name: {parameter_name: value, ... }, ... } for active components and free parameters.

property weight

A Weight object, able to assign significance weights to separate pixels or maps, given the model.

class hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy

Bases: *LocalStrategy*

Reduced chi-squared Local strategy of the SAMFire. Uses reduced chi-squared as the weight, and exponential decay as the decay function.

clean()

Purges the currently saved values.

plot(*fig=None*)

Plots the current marker in a flat image

Parameters

fig

[*hyperspy.api.signals.Signal2D* or *numpy.ndarray*] if an already plotted image, then updates. Otherwise creates a new one.

Returns

hyperspy.api.signals.Signal2D

The resulting 2D signal. If passed again, will be updated (computationally cheaper operation).

property radii

A tuple of ≥ 0 floats that show the “radii of relevance”

refresh(*overwrite*, *given_pixels=None*)

Refreshes the marker - recalculates with the current values from scratch.

Parameters

overwrite

[*bool*] If True, all but the *given_pixels* will be recalculated. Used when part of already calculated results has to be refreshed. If False, only use pixels with *marker == -scale* (by default -1) to propagate to pixels with *marker ≥ 0* . This allows “ignoring” pixels with *marker < -scale* (e.g. -2).

given_pixels

[*numpy.ndarray* of *bool*] Pixels with True value are assumed as correctly calculated.

remove()

Removes this strategy from its SAMFire

property samf

The SAMFire that owns this strategy.

update(*ind*, *isgood*)

Updates the database and marker with the given pixel results

Parameters

ind

[*tuple*] the index with new results

isgood

[*bool*] if the fit was successful.

values(*ind*)

Returns the current starting value estimates for the given pixel. Calculated as the weighted local average. Only returns components that are active, and parameters that are free.

Parameters

ind

[*tuple*] the index of the pixel of interest.

Returns

values

[*dict*] A dictionary of estimates, structured as {*component_name*: {*parameter_name*: *value*, ...}, ...} for active components and free parameters.

property weight

A Weight object, able to assign significance weights to separate pixels or maps, given the model.

SAMFire modules

The *samfire* module contains the following submodules:

fit_tests

Tests to check fit convergence when running SAMFire

global_strategies

Available global strategies to use in SAMFire

local_strategies

Available global strategies to use in SAMFire

SamfirePool

The parallel pool, customized to run SAMFire.

class `hyperspy.api.samfire.SamfirePool(**kwargs)`

Bases: *ParallelPool*

Creates and manages a pool of SAMFire workers. For based on ParallelPool - either creates processes using multiprocessing, or connects and sets up ipyparallel `load_balanced_view`.

Ipyparallel is managed directly, but multiprocessing pool is managed via three of queues:

- Shared by all (master and workers) for distributing “load-balanced” work.
- Shared by all (master and workers) for sending results back to the master
- Individual queues from master to each worker. For setting up and addressing individual workers in general. This one is checked with higher priority in workers.

Attributes

has_pool

[bool] Returns True if the pool is ready and set-up else False.

pool

[`ipyparallel.LoadBalancedView` or `multiprocessing.pool.Pool`] The pool object

ipython_kwargs

[dict] The dictionary with Ipyparallel connection arguments.

timeout

[float] Timeout for either pool when waiting for results

num_workers

[int] The number of workers actually created (may be less than requested, but can't be more)

timestep

[float] The timestep between “ticks” that the result queues are checked. Higher timestep means less frequent checking, which may reduce CPU load for difficult fits that take a long time to finish.

ping

[dict] If recorded, stores one-way trip time of each worker

pid

[dict] If available, stores the process-id of each worker

Creates a ParallelPool with additional methods for SAMFire. All arguments are passed to ParallelPool

add_jobs(*needed_number=None*)

Adds jobs to the job queue that is consumed by the workers.

Parameters

needed_number: {None, int}

The number of jobs to add. If None (default), adds *need_pixels*

collect_results(*timeout=None*)

Collects and parses all results, currently not processed due to being in the queue.

Parameters

timeout: {None, float}

the time to wait when collecting results. If None, the default timeout is used

property need_pixels

Returns the number of pixels that should be added to the processing queue. At most is equal to the number of workers.

parse(*value*)

Parse the value returned from the workers.

Parameters

value: tuple of the form (keyword, the_rest)

Keyword currently can be one of ['pong', 'Error', 'result']. For each of the keywords, "the_rest" is a tuple of different elements, but generally the first one is always the worker_id that the result came from. In particular:

- ('pong', (worker_id, pid, pong_time, optional_message_str))
- ('Error', (worker_id, error_message_string))
- ('result', (worker_id, pixel_index, result_dict, bool_if_result_converged))

ping_workers(*timeout=None*)

Ping the workers and record one-way trip time and the process_id pid of each worker if available.

Parameters

timeout: {None, float}

the time to wait when collecting results after sending out the ping. If None, the default timeout is used

prepare_workers(*samfire*)

Given SAMFire object, populate the workers with the required information. In case of multiprocessing, start worker listening to the queues.

Parameters

samfire

[*Samfire*] The SAMFire object that will be using the pool.

run()

Run the full process of adding jobs to the processing queue, listening to the results and updating SAMFire as needed. Stops when timed out or no pixels are left to run.

Run the full procedure until no more pixels are left to run in the SAMFire.

stop()

Stops the appropriate pool and (if ipyparallel) clears the memory and history.

update_parameters()

Updates various worker parameters.

Currently updates:

- Optional components (that can be switched off by the worker)
- Parameter boundaries
- Goodness test

23.8 hyperspy.api.signals

The Signal class and its specialized subclasses:

BaseSignal

For generic data with arbitrary `signal_dimension`. All other signal classes inherit from this one. It should only be used with none of the others is appropriated.

Signal1D

For generic data with `signal_dimension` equal 1, i.e. spectral data of n-dimensions. The signal is unbinned by default.

Signal2D

For generic data with `signal_dimension` equal 2, i.e. image data of n-dimensions. The signal is unbinned by default.

ComplexSignal

For generic complex data with arbitrary `signal_dimension`.

ComplexSignal1D

For generic complex data with `signal_dimension` equal 1, i.e. spectral data of n-dimensions. The signal is unbinned by default.

ComplexSignal2D

For generic complex data with `signal_dimension` equal 2, i.e. image data of n-dimensions. The signal is unbinned by default.

<i>BaseSignal</i>	Examples
<i>ComplexSignal</i>	General signal class for complex data.
<i>ComplexSignal1D</i>	Signal class for complex 1-dimensional data.
<i>ComplexSignal2D</i>	Signal class for complex 2-dimensional data.
<i>Signal1D</i>	General 1D signal class.
<i>Signal2D</i>	General 2D signal class.

23.8.1 BaseSignal

`class hyperspy.api.signals.BaseSignal(data, **kwargs)`

Bases: FancySlicing, MVA, MVATools

Examples

General signal created from a numpy or cupy array.

```
>>> data = np.ones((10, 10))
>>> s = hs.signals.BaseSignal(data)
```

Attributes

ragged

[bool] Whether the signal is ragged or not.

isig

Signal indexer/slicer.

inav

Navigation indexer/slicer.

metadata

[*hyperspy.misc.utils.DictionaryTreeBrowser*] The metadata of the signal.

original_metadata

[*hyperspy.misc.utils.DictionaryTreeBrowser*] The original metadata of the signal.

Create a signal instance.

Parameters

data

[*numpy.ndarray*] The signal data. It can be an array of any dimensions.

axes

[[dict/axes], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[dict, optional] A dictionary whose items are stored as attributes.

metadata

[dict, optional] A dictionary containing a set of parameters that will be stored in the metadata attribute. Some parameters might be mandatory in some cases.

original_metadata

[dict, optional] A dictionary containing a set of parameters that will be stored in the original_metadata attribute. It typically contains all the parameters that have been imported from the original data file.

ragged

[bool or None, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the attributes dictionary. If None, it does nothing. Default is None.

isig

inav

property T

The transpose of the signal, with signal and navigation spaces swapped. Enables calling `transpose()` with the default parameters as a property of a Signal.

add_gaussian_noise(*std*, *random_state=None*)

Add Gaussian noise to the data.

The operation is performed in-place (*i.e.* the data of the signal is modified). This method requires the signal to have a float data type, otherwise it will raise a `TypeError`.

Parameters

std

[float] The standard deviation of the Gaussian noise.

random_state

[None, int or `numpy.random.Generator`, default None] Seed for the random generator.

Notes

This method uses `numpy.random.normal()` (or `dask.array.random.normal()` for lazy signals) to generate the noise.

add_marker(*marker*, *plot_on_signal=True*, *plot_marker=True*, *permanent=False*, *plot_signal=True*, *render_figure=True*)

Add one or several markers to the signal or navigator plot and plot the signal, if not yet plotted (by default)

Parameters

marker

[*markers object* or iterable] The marker or iterable (list, tuple, ...) of markers to add. See the *Markers* section in the User Guide if you want to add a large number of markers as an iterable, since this will be much faster. For signals with navigation dimensions, the markers can be made to change for different navigation indices. See the examples for info.

plot_on_signal

[bool, default True] If True, add the marker to the signal. If False, add the marker to the navigator

plot_marker

[bool, default True] If True, plot the marker.

permanent

[bool, default True] If False, the marker will only appear in the current plot. If True, the marker will be added to the `metadata.Markers` list, and be plotted with `plot(plot_markers=True)`. If the signal is saved as a HyperSpy HDF5 file, the markers will be stored in the HDF5 signal and be restored when the file is loaded.

Examples

```
>>> im = hs.data.wave_image()
>>> m = hs.plot.markers.Rectangles(
...     offsets=[(1.0, 1.5)], widths=(0.5,), heights=(0.7,)
... )
>>> im.add_marker(m)
```

Add permanent marker:

```
>>> rng = np.random.default_rng(1)
>>> s = hs.signals.Signal2D(rng.random((100, 100)))
>>> marker = hs.plot.markers.Points(offsets=[(50, 60)])
>>> s.add_marker(marker, permanent=True, plot_marker=True)
```

Removing a permanent marker:

```
>>> rng = np.random.default_rng(1)
>>> s = hs.signals.Signal2D(rng.integers(10, size=(100, 100)))
>>> marker = hs.plot.markers.Points(offsets=[(10, 60)])
>>> marker.name = "point_marker"
>>> s.add_marker(marker, permanent=True)
>>> del s.metadata.Markers.point_marker
```

Adding many markers as a list:

```
>>> rng = np.random.default_rng(1)
>>> s = hs.signals.Signal2D(rng.integers(10, size=(100, 100)))
>>> marker_list = []
>>> for i in range(10):
...     marker = hs.plot.markers.Points(rng.random(2))
...     marker_list.append(marker)
>>> s.add_marker(marker_list, permanent=True)
```

add_poissonian_noise(*keep_dtype=True, random_state=None*)

Add Poissonian noise to the data.

This method works in-place. The resulting data type is `int64`. If this is different from the original data type then a warning is added to the log.

Parameters

keep_dtype

[`bool`, default `True`] If `True`, keep the original data type of the signal data. For example, if the data type was initially `'float64'`, the result of the operation (usually `'int64'`) will be converted to `'float64'`.

random_state

[`None`, `int` or `numpy.random.Generator`, default `None`] Seed for the random generator.

Notes

This method uses `numpy.random.poisson()` (or `dask.array.random.poisson()` for lazy signals) to generate the Poissonian noise.

apply_apodization(*window='hann', hann_order=None, tukey_alpha=0.5, inplace=False*)

Apply an apodization window to a Signal.

Parameters

window

[`str`, optional] Select between { 'hann' (default), 'hamming', or 'tukey' }

hann_order

[`None` or `int`, optional] Only used if `window='hann'` If integer *n* is provided, a Hann window of *n*-th order will be used. If `None`, a first order Hann window is used. Higher orders result in more homogeneous intensity distribution.

tukey_alpha

[`float`, optional] Only used if `window='tukey'` (default is 0.5). From the documentation of `scipy.signal.windows.tukey()`:

- Shape parameter of the Tukey window, representing the fraction of the window inside the cosine tapered region. If zero, the Tukey window is equivalent to a rectangular window. If one, the Tukey window is equivalent to a Hann window.

inplace

[`bool`, optional] If `True`, the apodization is applied in place, *i.e.* the signal data will be substituted by the apodized one (default is `False`).

Returns

out

[`BaseSignal` (or subclass), optional] If `inplace=False`, returns the apodized signal of the same type as the provided Signal.

Examples

```
>>> import hyperspy.api as hs
>>> wave = hs.data.wave_image()
>>> wave.apply_apodization('tukey', tukey_alpha=0.1).plot()
```

as_lazy(*copy_variance=True, copy_navigator=True, copy_learning_results=True*)

Create a copy of the given Signal as a `LazySignal`.

Parameters

copy_variance

[`bool`] Whether or not to copy the variance from the original Signal to the new lazy version. Default is `True`.

copy_navigator

[`bool`] Whether or not to copy the navigator from the original Signal to the new lazy version. Default is `True`.

copy_learning_results

[`bool`] Whether to copy the learning_results from the original signal to the new lazy version. Default is `True`.

Returns

res

[*LazySignal*] The same signal, converted to be lazy

as_signal1D(*spectral_axis*, *out=None*, *optimize=True*)

Return the Signal as a spectrum.

The chosen spectral axis is moved to the last index in the array and the data is made contiguous for efficient iteration over spectra. By default, the method ensures the data is stored optimally, hence often making a copy of the data. See [transpose\(\)](#) for a more general method with more options.

Parameters

spectral_axis

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

optimize

[*bool*] If *True*, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the [Transposing \(changing signal spaces\)](#) section of the HyperSpy user guide for more information. When operating on lazy signals, if *True*, the chunks are optimised for the new axes configuration.

See also:

[hyperspy.api.signals.BaseSignal.as_signal2D](#)

[hyperspy.api.signals.BaseSignal.transpose](#)

[hyperspy.api.transpose](#)

Examples

```
>>> img = hs.signals.Signal2D(np.ones((3, 4, 5, 6)))
>>> img
<Signal2D, title: , dimensions: (4, 3|6, 5)>
>>> img.as_signal1D(-1+1j)
<Signal1D, title: , dimensions: (6, 5, 4|3)>
>>> img.as_signal1D(0)
<Signal1D, title: , dimensions: (6, 5, 3|4)>
```

as_signal2D(*image_axes*, *out=None*, *optimize=True*)

Convert a signal to a *Signal2D*.

The chosen image axes are moved to the last indices in the array and the data is made contiguous for efficient iteration over images.

Parameters

image_axes

[*tuple* (of *int*, *str* or *DataAxis*)] Select the image axes. Note that the order of the axes matters and it is given in the “natural” i.e. X, Y, Z... order.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

optimize

[*bool*] If *True*, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if *True*, the chunks are optimised for the new axes configuration.

Raises**DataDimensionError**

When *data.ndim < 2*

See also:

[*hyperspy.api.signals.BaseSignal.as_signal1D*](#)

[*hyperspy.api.signals.BaseSignal.transpose*](#)

[*hyperspy.api.transpose*](#)

Examples

```
>>> s = hs.signals.Signal1D(np.ones((2, 3, 4, 5)))
>>> s
<Signal1D, title: , dimensions: (4, 3, 2|5)>
>>> s.as_signal2D((0, 1))
<Signal2D, title: , dimensions: (5, 2|4, 3)>
```

```
>>> s.to_signal2D((1, 2))
<Signal2D, title: , dimensions: (2, 5|4, 3)>
```

blind_source_separation(*number_of_components=None, algorithm='sklearn_fastica', diff_order=1, diff_axes=None, factors=None, comp_list=None, mask=None, on_loadings=False, reverse_component_criterion='factors', whiten_method='PCA', return_info=False, print_info=True, **kwargs*)

Apply blind source separation (BSS) to the result of a decomposition.

The results are stored in *self.learning_results*.

Read more in the *User Guide*.

Parameters**number_of_components**

[*int* or *None*] Number of principal components to pass to the BSS algorithm. If *None*, you must specify the *comp_list* argument.

algorithm

[{"sklearn_fastica" | "orthomax" | "FastICA" | "JADE" |

""CuBICA"" | ""TDSEP""} or object, default "sklearn_fastica"

The BSS algorithm to use. If *algorithm* is an object, it must implement a

`fit_transform()` method or `fit()` and `transform()` methods, in the same manner as a scikit-learn estimator.

diff_order

[`int`, default 1] Sometimes it is convenient to perform the BSS on the derivative of the signal. If `diff_order` is 0, the signal is not differentiated.

diff_axes

[`None`, `list` of `int`, `list` of `str`]

- If `None` and `on_loadings` is `False`, when `diff_order` is greater than 1 and `signal_dimension` is greater than 1, the differences are calculated across all signal axes
- If `None` and `on_loadings` is `True`, when `diff_order` is greater than 1 and `navigation_dimension` is greater than 1, the differences are calculated across all navigation axes
- Otherwise the axes can be specified in a list.

factors

[`BaseSignal` or `numpy.ndarray`] Factors to decompose. If `None`, the BSS is performed on the factors of a previous decomposition. If a `Signal` instance, the navigation dimension must be 1 and the size greater than 1.

comp_list

[`None` or `list` or `numpy.ndarray`] Choose the components to apply BSS to. Unlike `number_of_components`, this argument permits non-contiguous components.

mask

[`BaseSignal` or subclass] If not `None`, the signal locations marked as `True` are masked. The mask shape must be equal to the signal shape (navigation shape) when `on_loadings` is `False` (`True`).

on_loadings

[`bool`, default `False`] If `True`, perform the BSS on the loadings of a previous decomposition, otherwise, perform the BSS on the factors.

reverse_component_criterion

[{"factors", "loadings"}, default "factors"] Use either the factors or the loadings to determine if the component needs to be reversed.

whiten_method

[{"PCA" | "ZCA"} or `None`, default "PCA"] How to whiten the data prior to blind source separation. If `None`, no whitening is applied. See `whiten_data()` for more details.

return_info: bool, default False

The result of the decomposition is stored internally. However, some algorithms generate some extra information that is not stored. If `True`, return any extra information if available. In the case of `sklearn.decomposition` objects, this includes the `sklearn` Estimator object.

print_info

[`bool`, default `True`] If `True`, print information about the decomposition being performed. In the case of `sklearn.decomposition` objects, this includes the values of all arguments of the chosen `sklearn` algorithm.

****kwargs**

[`dict`] Any keyword arguments are passed to the BSS algorithm.

Returns

`None` or subclass of `sklearn.base.BaseEstimator`

If True and ‘algorithm’ is an sklearn Estimator, returns the Estimator object.

See also:

[plot_bss_factors](#), [plot_bss_loadings](#), [plot_bss_results](#)

change_dtype(*dtype*, *rechunk=False*)

Change the data type of a Signal.

Parameters

dtype

[[str](#) or [numpy.dtype](#)] Typecode string or data-type to which the Signal’s data array is cast. In addition to all the standard numpy [Data type objects \(dtype\)](#), HyperSpy supports four extra dtypes for RGB images: ‘rgb8’, ‘rgba8’, ‘rgb16’, and ‘rgba16’. Changing from and to any [rgb\(a\) dtype](#) is more constrained than most other [dtype](#) conversions. To change to an [rgb\(a\) dtype](#), the *signal_dimension* must be 1, and its size should be 3 (for [rgb](#)) or 4 (for [rgba](#)) [dtypes](#). The original [dtype](#) should be [uint8](#) or [uint16](#) if converting to [rgb\(a\)8](#) or [rgb\(a\)16](#), and the *navigation_dimension* should be at least 2. After conversion, the *signal_dimension* becomes 2. The [dtype](#) of images with original [dtype](#) [rgb\(a\)8](#) or [rgb\(a\)16](#) can only be changed to [uint8](#) or [uint16](#), and the *signal_dimension* becomes 1.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

Examples

```
>>> s = hs.signals.Signal1D([1, 2, 3, 4, 5])
>>> s.data
array([1, 2, 3, 4, 5])
>>> s.change_dtype('float')
>>> s.data
array([1., 2., 3., 4., 5.])
```

cluster_analysis(*cluster_source*, *source_for_centers=None*, *preprocessing=None*, *preprocessing_kwargs=None*, *number_of_components=None*, *navigation_mask=None*, *signal_mask=None*, *algorithm=None*, *return_info=False*, ***kwargs*)

Cluster analysis of a signal or decomposition results of a signal Results are stored in *learning_results*.

Parameters

cluster_source

[[str](#) {“bss” | “decomposition” | “signal”} or [BaseSignal](#)] If “bss” the blind source separation results are used If “decomposition” the decomposition results are used if “signal” the signal data is used Note that using the signal or [BaseSignal](#) can be memory intensive and is only recommended if the Signal dimension is small [BaseSignal](#) must have the same navigation dimensions as the signal.

source_for_centers

[[None](#), [str](#) {“decomposition” | “bss” | “signal”} or [BaseSignal](#)] default : [None](#) If [None](#) the *cluster_source* is used If “bss” the blind source separation results are used

If “decomposition” the decomposition results are used if “signal” the signal data is used BaseSignal must have the same navigation dimensions as the signal.

preprocessing

[`str` {"standard" | "norm" | "minmax"}, `None` or `object`] default: 'norm' Preprocessing the data before cluster analysis requires preprocessing the data to be clustered to similar scales. Standard preprocessing adjusts each feature to have uniform variation. Norm preprocessing adjusts treats the set of features like a vector and each measurement is scaled to length 1. You can also pass one of the scikit-learn preprocessing `scale_method = import sklearn.preprocessing.StandadScaler()` preprocessing = `scale_method` See preprocessing methods in scikit-learn preprocessing for further details. If `object`, must be `sklearn.preprocessing`-like.

preprocessing_kwargs

[`dict` or `None`, default `None`] Additional parameters passed to the supported sklearn preprocessing methods. See `sklearn.preprocessing` scaling methods for further details

number_of_components

[`int`, default `None`] If you are getting the cluster centers using the decomposition results (`cluster_source_for_centers="decomposition"`) you can define how many components to use. If set to `None` the method uses the estimate of significant components found in the decomposition step using the elbow method and stored in the `learning_results.number_significant_components` attribute. This applies to both `bss` and decomposition results.

navigation_mask

[`numpy.ndarray` of `bool`] The navigation locations marked as `True` are not used.

signal_mask

[`numpy.ndarray` of `bool`] The signal locations marked as `True` are not used in the clustering for “signal” or Signals supplied as cluster source. This is not applied to decomposition results or `source_for_centers` (as it may be a different shape to the cluster source)

algorithm

[{"kmeans" | "agglomerative" | "minibatchkmeans" | "spectralclustering"}]
See scikit-learn documentation. Default “kmeans”

return_info

[`bool`, default `False`] The result of the cluster analysis is stored internally. However, the cluster class used contain a number of attributes. If `True` (the default is `False`) return the cluster object so the attributes can be accessed.

****kwargs**

[`dict`] Additional parameters passed to the clustering class for initialization. For example, in case of the “kmeans” algorithm, `n_init` can be used to define the number of times the algorithm is restarted to optimize results.

Returns**None or object**

If 'return_info' is `True` returns the Scikit-learn cluster object used for clustering. Useful if you wish to examine inertia or other outputs.

Other Parameters**int**

Number of clusters to find using the one of the pre-defined methods “kmeans”, “agglomerative”, “minibatchkmeans”, “spectralclustering” See `sklearn.cluster` for details

See also:

*estimate_number_of_clusters, get_cluster_labels, get_cluster_signals
get_cluster_distances, plot_cluster_metric, plot_cluster_results
plot_cluster_signals, plot_cluster_labels*

copy()

Return a “shallow copy” of this Signal using the standard library’s `copy()` function. Note: this will return a copy of the signal, but it will not duplicate the underlying data in memory, and both Signals will reference the same data.

See also:

deepcopy()

crop(*axis, start=None, end=None, convert_units=False*)

Crops the data in a given axis. The range is given in pixels.

Parameters

axis

[`int` or `str`] Specify the data axis in which to perform the cropping operation. The axis can be specified using the index of the axis in *axes_manager* or the axis name.

start

[`int`, `float`, or `None`] The beginning of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

end

[`int`, `float`, or `None`] The end of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

convert_units

[`bool`] Default is `False`. If `True`, convert the units using the *convert_units()* method of the *AxesManager*. If `False`, does nothing.

property data

The underlying data structure as a `numpy.ndarray` (or `dask.array.Array`, if the Signal is lazy).

decomposition(*normalize_poissonian_noise=False, algorithm='SVD', output_dimension=None, centre=None, auto_transpose=True, navigation_mask=None, signal_mask=None, var_array=None, var_func=None, reproject=None, return_info=False, print_info=True, svd_solver='auto', copy=True, **kwargs*)

Apply a decomposition to a dataset with a choice of algorithms.

The results are stored in `self.learning_results`.

Read more in the *User Guide*.

Parameters

normalize_poissonian_noise

[`bool`, default `False`] If `True`, scale the signal to normalize Poissonian noise using the approach described in [*].

algorithm

[`str` {"SVD", "MLPCA", "sklearn_pca", "NMF", "sparse_pca",}]

`""mini_batch_sparse_pca""`, `""RPCA""`, `""ORPCA""`, `""ORNMF""`} or object, default `""SVD""`

The decomposition algorithm to use. If algorithm is an object, it must implement a `fit_transform()` method or `fit()` and `transform()` methods, in the same manner as a scikit-learn estimator. For cupy arrays, only “SVD” is supported.

output_dimension

[`None` or `int`] Number of components to keep/calculate. Default is `None`, i.e. `min(data.shape)`.

centre

[`None` or `str` {"navigation", "signal"}], default `None`]

- If `None`, the data is not centered prior to decomposition.
- If “navigation”, the data is centered along the navigation axis. Only used by the “SVD” algorithm.
- If “signal”, the data is centered along the signal axis. Only used by the “SVD” algorithm.

auto_transpose

[`bool`, default `True`] If `True`, automatically transposes the data to boost performance. Only used by the “SVD” algorithm.

navigation_mask

[`numpy.ndarray` or `BaseSignal`] The navigation locations marked as `True` are not used in the decomposition.

signal_mask

[`numpy.ndarray` or `BaseSignal`] The signal locations marked as `True` are not used in the decomposition.

var_array

[`numpy.ndarray`] Array of variance for the maximum likelihood PCA algorithm. Only used by the “MLPCA” algorithm.

var_func

[`None`, `callable()` or `numpy.ndarray`, default `None`] If `None`, ignored. If callable, applies the function to the data to obtain `var_array`. Only used by the “MLPCA” algorithm. If `numpy` array, creates `var_array` by applying a polynomial function defined by the array of coefficients to the data. Only used by the “MLPCA” algorithm.

reproject

[`None` or `str` {"signal", "navigation", "both"}], default `None`] If not `None`, the results of the decomposition will be projected in the selected masked area.

return_info: bool, default False

The result of the decomposition is stored internally. However, some algorithms generate some extra information that is not stored. If `True`, return any extra information if available. In the case of `sklearn.decomposition` objects, this includes the `sklearn` Estimator object.

print_info

[`bool`, default `True`] If `True`, print information about the decomposition being performed. In the case of `sklearn.decomposition` objects, this includes the values of all arguments of the chosen `sklearn` algorithm.

svd_solver

[{"auto", "full", "arpack", "randomized"}], default “auto”]

- If “auto”: the solver is selected by a default policy based on `data.shape` and `output_dimension`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more

efficient "randomized" method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

- If "full": run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing
- If "arpack": use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It strictly requires $0 < \text{output_dimension} < \min(\text{data.shape})$
- If "randomized": use truncated SVD, call `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

For cupy arrays, only "full" is supported.

copy

[bool, default True]

- If True, stores a copy of the data before any pre-treatments such as normalization in `s._data_before_treatments`. The original data can then be restored by calling `s.undo_treatments()`.
- If False, no copy is made. This can be beneficial for memory usage, but care must be taken since data will be overwritten.

**kwargs

[dict] Any keyword arguments are passed to the decomposition algorithm.

Returns

tuple of `numpy.ndarray` or `sklearn.base.BaseEstimator` or None

- If True and 'algorithm' in ['RPCA', 'ORPCA', 'ORNMF'], returns the low-rank (X) and sparse (E) matrices from robust PCA/NMF.
- If True and 'algorithm' is an sklearn Estimator, returns the Estimator object.
- Otherwise, returns None

See also:

[plot_decomposition_factors](#), [plot_decomposition_loadings](#)
[plot_decomposition_results](#), [plot_explained_variance_ratio](#)

References

deepcopy()

Return a "deep copy" of this Signal using the standard library's `deepcopy()` function. Note: this means the underlying data structure will be duplicated in memory.

See also:

[copy\(\)](#)

derivative(axis, order=1, out=None, **kwargs)

Calculate the numerical derivative along the given axis, with respect to the calibrated units of that axis.

For a function $y = f(x)$ and two consecutive values x_1 and x_2 :

$$\frac{df(x)}{dx} = \frac{y(x_2) - y(x_1)}{x_2 - x_1}$$

Parameters

axis

[[int](#), [str](#), or [DataAxis](#)] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

order: int

The order of the derivative.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

****kwargs**

[[dict](#)] All extra keyword arguments are passed to `numpy.gradient()`

Returns

[BaseSignal](#)

Note that the size of the data on the given axis decreases by the given order. *i.e.* if axis is "x" and order is 2, if the x dimension is N, then der's x dimension is N - 2.

See also:

[integrate1D](#), [integrate_simpson](#)

Notes

This function uses `numpy.gradient` to perform the derivative. See its documentation for implementation details.

diff(*axis*, *order=1*, *out=None*, *rechunk=False*)

Returns a signal with the n-th order discrete difference along given axis. *i.e.* it calculates the difference between consecutive values in the given axis: $out[n] = a[n+1] - a[n]$. See `numpy.diff()` for more details.

Parameters

axis

[[int](#), [str](#), or [DataAxis](#)] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

order

[[int](#)] The order of the discrete difference.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

Returns

[BaseSignal](#) or [None](#)

Note that the size of the data on the given axis decreases by the given order. *i.e.* if axis is "x" and order is 2, the x dimension is N, der's x dimension is N - 2.

See also:

`hyperspy.api.signals.BaseSignal.derivative`
`hyperspy.api.signals.BaseSignal.integrate1D`
`hyperspy.api.signals.BaseSignal.integrate_simpson`

Notes

If you intend to calculate the numerical derivative, please use the proper `derivative()` function instead. To avoid erroneous misuse of the `diff` function as derivative, it raises an error when working with a non-uniform axis.

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.diff(0)
<BaseSignal, title: , dimensions: (|1023, 64, 64)>
```

`estimate_elbow_position(explained_variance_ratio=None, log=True, max_points=20)`

Estimate the elbow position of a scree plot curve.

Used to estimate the number of significant components in a PCA variance ratio plot or other “elbow” type curves.

Find a line between first and last point on the scree plot. With a classic elbow scree plot, this line more or less defines a triangle. The elbow should be the point which is the furthest distance from this line. For more details, see [1].

Parameters

explained_variance_ratio

[{None, numpy array}] Explained variance ratio values that form the scree plot. If None, uses the `explained_variance_ratio` array stored in `s.learning_results`, so a decomposition must have been performed first.

max_points

[int] Maximum number of points to consider in the calculation.

Returns

int

The index of the elbow position in the input array. Due to zero-based indexing, the number of significant components is `elbow_position + 1`.

See also:

`get_explained_variance_ratio`, `plot_explained_variance_ratio`

References

[1]

estimate_number_of_clusters(*cluster_source*, *max_clusters=10*, *preprocessing=None*, *preprocessing_kwargs=None*, *number_of_components=None*, *navigation_mask=None*, *signal_mask=None*, *algorithm=None*, *metric='gap'*, *n_ref=4*, *show_progressbar=None*, ***kwargs*)

Performs cluster analysis of a signal for cluster sizes ranging from `n_clusters = 2` to `max_clusters` (default 12) Note that this can be a slow process for large datasets so please consider reducing `max_clusters` in this case. For each cluster it evaluates the silhouette score which is a metric of how well separated the clusters are. Maximima or peaks in the scores indicate good choices for cluster sizes.

Parameters

cluster_source

[`str` {"bss", "decomposition", "signal"} or `BaseSignal`] If "bss" the blind source separation results are used If "decomposition" the decomposition results are used if "signal" the signal data is used Note that using the signal can be memory intensive and is only recommended if the Signal dimension is small. Input Signal must have the same navigation dimensions as the signal instance.

max_clusters

[`int`, default 10] Max number of clusters to use. The method will scan from 2 to `max_clusters`.

preprocessing

[`str` {"standard", "norm", "minmax"} or `object`] default: 'norm' Preprocessing the data before cluster analysis requires preprocessing the data to be clustered to similar scales. Standard preprocessing adjusts each feature to have uniform variation. Norm preprocessing adjusts treats the set of features like a vector and each measurement is scaled to length 1. You can also pass an instance of a sklearn preprocessing module. See preprocessing methods in scikit-learn preprocessing for further details. If `object`, must be `sklearn.preprocessing`-like.

preprocessing_kwargs

[`dict` or `None`, default `None`] Additional parameters passed to the cluster preprocessing algorithm. See `sklearn.preprocessing` preprocessing methods for further details

number_of_components

[`int`, default `None`] If you are getting the cluster centers using the decomposition results (`cluster_source_for_centers="decomposition"`) you can define how many PCA components to use. If set to `None` the method uses the estimate of significant components found in the decomposition step using the elbow method and stored in the `learning_results.number_significant_components` attribute.

navigation_mask

[`bool numpy array`, default][`None`] The navigation locations marked as `True` are not used in the clustering.

signal_mask

[`numpy.ndarray` of `bool`, default `None`] The signal locations marked as `True` are not used in the clustering. Applies to "signal" or `Signal` cluster sources only.

metric

[{"elbow" | "silhouette" | "gap"}, default 'gap'] Use distance, silhouette analysis or gap statistics to estimate the optimal number of clusters. Gap is believed to be, overall, the best metric but it's also the slowest. Elbow measures the distances between points in each cluster as an estimate of how well grouped they are and is the fastest metric. For elbow

the optimal k is the knee or elbow point. For gap the optimal k is the first k $\text{gap}(k) \geq \text{gap}(k+1) - \text{std_error}$. For silhouette the optimal k will be one of the “maxima” found with this method

n_ref

[`int`, default 4] Number of references to use in gap statistics method. Gap statistics compares the results from clustering the data to clustering uniformly distributed data. As clustering has a random variation it is typically averaged `n_ref` times to get an statistical average.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

****kwargs**

[`dict`] Parameters passed to the clustering algorithm.

Returns**int**

Estimate of the best cluster size.

Other Parameters**n_clusters**

[`int`] Number of clusters to find using the one of the pre-defined methods “kmeans”, “agglomerative”, “minibatchkmeans”, “spectralclustering”. See `sklearn.cluster` for details

See also:

[cluster_analysis](#), [get_cluster_labels](#), [get_cluster_signals](#),
[get_cluster_distances](#), [plot_cluster_metric](#), [plot_cluster_results](#),
[plot_cluster_signals](#), [plot_cluster_labels](#)

estimate_poissonian_noise_variance(*expected_value=None, gain_factor=None, gain_offset=None, correlation_factor=None*)

Estimate the Poissonian noise variance of the signal.

The variance is stored in the `metadata.Signal.Noise_properties.variance` attribute.

The Poissonian noise variance is equal to the expected value. With the default arguments, this method simply sets the variance attribute to the given *expected_value*. However, more generally (although then the noise is not strictly Poissonian), the variance may be proportional to the expected value. Moreover, when the noise is a mixture of white (Gaussian) and Poissonian noise, the variance is described by the following linear model:

$$\text{Var}[X] = (a * E[X] + b) * c$$

Where *a* is the *gain_factor*, *b* is the *gain_offset* (the Gaussian noise variance) and *c* the *correlation_factor*. The correlation factor accounts for correlation of adjacent signal elements that can be modeled as a convolution with a Gaussian point spread function.

Parameters**expected_value**

[`None` or `BaseSignal` (or subclass)] If `None`, the signal data is taken as the expected value. Note that this may be inaccurate where the value of *data* is small.

gain_factor

[None or float] a in the above equation. Must be positive. If None, take the value from `metadata.Signal.Noise_properties.Variance_linear_model` if defined. Otherwise, suppose pure Poissonian noise (i.e. `gain_factor=1`). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

gain_offset

[None or float] b in the above equation. Must be positive. If None, take the value from `metadata.Signal.Noise_properties.Variance_linear_model` if defined. Otherwise, suppose pure Poissonian noise (i.e. `gain_offset=0`). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

correlation_factor

[None or float] c in the above equation. Must be positive. If None, take the value from `metadata.Signal.Noise_properties.Variance_linear_model` if defined. Otherwise, suppose pure Poissonian noise (i.e. `correlation_factor=1`). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

```
export_bss_results(comp_ids=None, folder=None, calibrate=True, multiple_files=True,
                    save_figures=False, factor_prefix='bss_factor', factor_format='hspy',
                    loading_prefix='bss_loading', loading_format='hspy', comp_label=None,
                    cmap=<matplotlib.colors.LinearSegmentedColormap object>, same_window=False,
                    no_nans=True, per_row=3, save_figures_format='png')
```

Export results from ICA to any of the supported formats.

Parameters**comp_ids**

[None, int or list of int] If None, returns all components/loadings. If an int, returns components/loadings with ids from 0 to the given value. If a list of ints, returns components/loadings with ids provided in the given list.

folder

[str or None] The path to the folder where the file will be saved. If None the current folder is used by default.

factor_prefix

[str] The prefix that any exported filenames for factors/components begin with

factor_format

[str] The extension of the format that you wish to save the factors to. Default is 'hspy'. See `loading_format` for more details.

loading_prefix

[str] The prefix that any exported filenames for factors/components begin with

loading_format

[str] The extension of the format that you wish to save to. default is 'hspy'. The format determines the kind of output:

- For image formats ('tif', 'png', 'jpg', etc.), plots are created using the plotting flags as below, and saved at 600 dpi. One plot is saved per loading.
- For multidimensional formats ('rpl', 'hspy'), arrays are saved in single files. All loadings are contained in the one file.
- For spectral formats ('msa'), each loading is saved to a separate file.

multiple_files

[bool] If True, one file will be created for each factor and loading. Otherwise, only two

files will be created, one for the factors and another for the loadings. The default value can be chosen in the preferences.

save_figures

[**bool**] If **True**, the same figures that are obtained when using the plot methods will be saved with 600 dpi resolution

Other Parameters

calibrate

[**bool**] If **True**, calibrates plots where calibration is available from the `axes_manager`. If **False**, plots are in pixels/channels.

same_window

[**bool**] If **True**, plots each factor to the same window.

comp_label

[**str**] the label that is either the plot title (if plotting in separate windows) or the label in the legend (if plotting in the same window)

cmap

[**Colormap**] The colormap used for images, such as factors, loadings, or for peak characteristics. Default is the matplotlib gray colormap (`plt.cm.gray`).

per_row

[**int**] The number of plots in each row, when the `same_window` parameter is **True**.

save_figures_format

[**str**] The image format extension.

See also:

[*get_bss_factors*](#), [*get_bss_loadings*](#)

Notes

The following parameters are only used when `save_figures = True`

```
export_cluster_results(cluster_ids=None, folder=None, calibrate=True, center_prefix='cluster_center',
                        center_format='hspy', membership_prefix='cluster_label',
                        membership_format='hspy', comp_label=None,
                        cmap=<matplotlib.colors.LinearSegmentedColormap object>,
                        same_window=False, multiple_files=True, no_nans=True, per_row=3,
                        save_figures=False, save_figures_format='png')
```

Export results from a cluster analysis to any of the supported formats.

Parameters

cluster_ids

[**None**, **int** or **list** of **int**] if **None**, returns all clusters/centers. if **int**, returns clusters/centers with ids from 0 to given int. if **list** of ints, returns clusters/centers with ids in given list.

folder

[**str** or **None**] The path to the folder where the file will be saved. If **None** the current folder is used by default.

center_prefix

[**str**] The prefix that any exported filenames for cluster centers begin with

center_format

[*str*] The extension of the format that you wish to save to. Default is “hspy”. See *loading format* for more details.

label_prefix

[*str*] The prefix that any exported filenames for cluster labels begin with

label_format

[*str*] The extension of the format that you wish to save to. default is “hspy”. The format determines the kind of output.

- **For image formats ('tif', 'png', 'jpg', etc.),**
plots are created using the plotting flags as below, and saved at 600 dpi. One plot is saved per loading.
- **For multidimensional formats ('rpl', 'hspy'), arrays**
are saved in single files. All loadings are contained in the one file.
- **For spectral formats ('msa'), each loading is saved to a**
separate file.

multiple_files

[*bool*, default *False*] If True, on exporting a file per center will be created. Otherwise only two files will be created, one for the centers and another for the membership. The default value can be chosen in the preferences.

save_figures

[*bool*, default *False*] If True the same figures that are obtained when using the plot methods will be saved with 600 dpi resolution

Other Parameters

These parameters are plotting options and only used when
`save_figures=True`.

calibrate

[*bool*] if True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

same_window

[*bool*] if True, plots each factor to the same window.

comp_label

[*str*] The label that is either the plot title (if plotting in separate windows) or the label in the legend (if plotting in the same window)

cmap

[*matplotlib.colors.Colormap*] The colormap used for the factor image, or for peak characteristics, the colormap used for the scatter plot of some peak characteristic.

per_row

[*int*] the number of plots in each row, when the same_window=True.

save_figures_format

[*str*] The image format extension.

See also:

get_cluster_signals

get_cluster_labels

```
export_decomposition_results(comp_ids=None, folder=None, calibrate=True, factor_prefix='factor',
                             factor_format='hspy', loading_prefix='loading', loading_format='hspy',
                             comp_label=None,
                             cmap=<matplotlib.colors.LinearSegmentedColormap object>,
                             same_window=False, multiple_files=True, no_nans=True, per_row=3,
                             save_figures=False, save_figures_format='png')
```

Export results from a decomposition to any of the supported formats.

Parameters

comp_ids

[[None](#), [int](#) or [list](#) of [int](#)] If [None](#), returns all components/loadings. If an [int](#), returns components/loadings with ids from 0 to the given value. If a list of [ints](#), returns components/loadings with ids provided in the given list.

folder

[[str](#) or [None](#)] The path to the folder where the file will be saved. If [None](#), the current folder is used by default.

factor_prefix

[[str](#)] The prefix that any exported filenames for factors/components begin with

factor_format

[[str](#)] The extension of the format that you wish to save the factors to. Default is 'hspy'. See *loading_format* for more details.

loading_prefix

[[str](#)] The prefix that any exported filenames for factors/components begin with

loading_format

[[str](#)] The extension of the format that you wish to save to. default is 'hspy'. The format determines the kind of output:

- For image formats ('tif', 'png', 'jpg', etc.), plots are created using the plotting flags as below, and saved at 600 dpi. One plot is saved per loading.
- For multidimensional formats ('rpl', 'hspy'), arrays are saved in single files. All loadings are contained in the one file.
- For spectral formats ('msa'), each loading is saved to a separate file.

multiple_files

[[bool](#)] If [True](#), one file will be created for each factor and loading. Otherwise, only two files will be created, one for the factors and another for the loadings. The default value can be chosen in the preferences.

save_figures

[[bool](#)] If [True](#) the same figures that are obtained when using the plot methods will be saved with 600 dpi resolution

Other Parameters

calibrate

[[bool](#)] If [True](#), calibrates plots where calibration is available from the `axes_manager`. If [False](#), plots are in pixels/channels.

same_window

[[bool](#)] If [True](#), plots each factor to the same window.

comp_label

[[str](#)] the label that is either the plot title (if plotting in separate windows) or the label in the legend (if plotting in the same window)

cmap

[[Colormap](#)] The colormap used for images, such as factors, loadings, or for peak characteristics. Default is the matplotlib gray colormap (`plt.cm.gray`).

per_row

[[int](#)] The number of plots in each row, when the *same_window* parameter is True.

save_figures_format

[[str](#)] The image format extension.

See also:

[get_decomposition_factors](#), [get_decomposition_loadings](#)

Notes

The following parameters are only used when `save_figures = True`

fft(*shift=False, apodization=False, real_fft_only=False, **kwargs*)

Compute the discrete Fourier Transform.

This function computes the discrete Fourier Transform over the signal axes by means of the Fast Fourier Transform (FFT) as implemented in numpy.

Parameters**shift**

[[bool](#), optional] If True, the origin of FFT will be shifted to the centre (default is False).

apodization

[[bool](#) or [str](#)] Apply an [apodization window](#) before calculating the FFT in order to suppress streaks. Valid string values are { 'hann' or 'hamming' or 'tukey' } If True or 'hann', applies a Hann window. If 'hamming' or 'tukey', applies Hamming or Tukey windows, respectively (default is False).

real_fft_only

[[bool](#), default False] If True and data is real-valued, uses `numpy.fft.rfftn()` instead of `numpy.fft.fftn()`

****kwargs**

[[dict](#)] other keyword arguments are described in `numpy.fft.fftn()`

Returns

s

[[ComplexSignal](#)] A Signal containing the result of the FFT algorithm

Raises**[NotImplementedError](#)**

If performing FFT along a non-uniform axis.

Notes

Requires a uniform axis. For further information see the documentation of `numpy.fft.fftn()`

Examples

```
>>> import skimage
>>> im = hs.signals.Signal2D(skimage.data.camera())
>>> im.fft()
<ComplexSignal2D, title: FFT of , dimensions: (|512, 512)>
```

```
>>> # Use following to plot power spectrum of `im`:
>>> im.fft(shift=True, apodization=True).plot(power_spectrum=True)
```

fold()

If the signal was previously unfolded, fold it back

get_bss_factors()

Return the blind source separation factors.

Returns

BaseSignal (or subclass)

See also:

[*get_bss_loadings*](#), [*export_bss_results*](#)

get_bss_loadings()

Return the blind source separation loadings.

Returns

BaseSignal (or subclass)

See also:

[*get_bss_factors*](#), [*export_bss_results*](#)

get_bss_model(*components=None, chunks='auto'*)

Generate model with the selected number of independent components.

Parameters

components

[None, int or list of int, default None] If None, rebuilds signal instance from all components If int, rebuilds signal instance from components in range 0-given int If list of ints, rebuilds signal instance from only components in given list

Returns

BaseSignal or subclass

A model built from the given components.

get_cluster_distances()

Euclidian distances to the centroid of each cluster

Returns

signal

Hyperspy signal of cluster distances

See also:

[*get_cluster_signals*](#)

get_cluster_labels(*merged=False*)

Return cluster labels as a Signal.

Parameters**merged**

[bool, default `False`] If `False` the cluster label signal has a navigation axes of length `number_of_clusters` and the signal along the the navigation direction is binary - 0 the point is not in the cluster, 1 it is included. If `True`, the cluster labels are merged (no navigation axes). The value of the signal at any point will be between -1 and the number of clusters. -1 represents the points that were masked for cluster analysis if any.

Returns**BaseSignal**

The cluster labels

See also:

[*get_cluster_signals*](#)

get_cluster_signals(*signal='mean'*)

Return the cluster centers as a Signal.

Parameters**signal**

[{"mean", "sum", "centroid"}, optional] If "mean" or "sum" return the mean signal or sum respectively over each cluster. If "centroid", returns the signals closest to the centroid.

See also:

[*get_cluster_labels*](#)

get_current_signal(*auto_title=True, auto_filename=True, as_numpy=False*)

Returns the data at the current coordinates as a *BaseSignal* subclass.

The signal subclass is the same as that of the current object. All the axes navigation attributes are set to `False`.

Parameters**auto_title**

[bool] If `True`, the current indices (in parentheses) are appended to the title, separated by a space, otherwise the title of the signal is used unchanged.

auto_filename

[bool] If `True` and *tmp_parameters.filename* is defined (which is always the case when the Signal has been read from a file), the filename stored in the metadata is modified by appending an underscore and the current indices in parentheses.

as_numpy

[`bool` or `None`] Only with cupy array. If True, return the current signal as numpy array, otherwise return as cupy array.

Returns**cs**

[*BaseSignal* (or subclass)] The data at the current coordinates as a Signal

Examples

```
>>> im = hs.signals.Signal2D(np.zeros((2, 3, 32, 32)))
>>> im
<Signal2D, title: , dimensions: (3, 2|32, 32)>
>>> im.axes_manager.indices = (2, 1)
>>> im.get_current_signal()
<Signal2D, title: (2, 1), dimensions: (|32, 32)>
```

get_decomposition_factors()

Return the decomposition factors.

Returns**signal**

[*BaseSignal* (or subclass)]

See also:

[*get_decomposition_loadings*](#), [*export_decomposition_results*](#)

get_decomposition_loadings()

Return the decomposition loadings.

Returns**signal**

[*BaseSignal* (or subclass)]

See also:

[*get_decomposition_factors*](#), [*export_decomposition_results*](#)

get_decomposition_model (*components=None*)

Generate model with the selected number of principal components.

Parameters**components**

[`None`, `int` or `list` of `int`, default `None`]

- If `None`, rebuilds signal instance from all components
- If `int`, rebuilds signal instance from components in range 0-given int
- If list of ints, rebuilds signal instance from only components in given list

Returns***BaseSignal* or subclass**

A model built from the given components.

get_dimensions_from_data()

Get the dimension parameters from the Signal's underlying data. Useful when the data structure was externally modified, or when the spectrum image was not loaded from a file

get_explained_variance_ratio()

Return explained variance ratio of the PCA components as a SignalID.

Read more in the *User Guide*.

Returns

s

[*SignalID*] Explained variance ratio.

See also:

decomposition, *plot_explained_variance_ratio*
get_decomposition_loadings, *get_decomposition_factors*

get_histogram(bins='fd', range_bins=None, max_num_bins=250, out=None, **kwargs)

Return a histogram of the signal data.

More sophisticated algorithms for determining the bins can be used by passing a string as the `bins` argument. Other than the 'blocks' and 'knuth' methods, the available algorithms are the same as `numpy.histogram()`.

Note: The lazy version of the algorithm only supports "scott" and "fd" as a string argument for `bins`.

Parameters**bins**

[`int` or `sequence` of `float` or `str`, default "fd"] If `bins` is an `int`, it defines the number of equal-width bins in the given range. If `bins` is a `sequence`, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If `bins` is a string from the list below, will use the method chosen to calculate the optimal bin width and consequently the number of bins (see Notes for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the 'auto' option is suggested. Weighted data is not supported for automated bin size selection.

'auto'

Maximum of the 'sturges' and 'fd' estimators. Provides good all around performance.

'fd' (Freedman Diaconis Estimator)

Robust (resilient to outliers) estimator that takes into account data variability and data size.

'doane'

An improved version of Sturges' estimator that works better with non-normal datasets.

'scott'

Less robust estimator that that takes into account data variability and data size.

'stone'

Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott's rule.

‘rice’

Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

‘sturges’

R’s default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.

‘sqrt’

Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.

‘knuth’

Knuth’s rule is a fixed-width, Bayesian approach to determining the optimal bin width of a histogram.

‘blocks’

Determination of optimal adaptive-width histogram bins using the Bayesian Blocks algorithm.

range_bins

[[tuple](#) or [None](#), optional] the minimum and maximum range for the histogram. If `range_bins` is [None](#), (`x.min()`, `x.max()`) will be used.

max_num_bins

[[int](#), default 250] When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a `MemoryError` being raised by `numpy.histogram()`.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new `Signal` is created with the result of the operation and returned (default). If a `Signal` is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

****kwargs**

other keyword arguments (weight and density) are described in `numpy.histogram()`.

Returns**hist_spec**

[[Signal1D](#)] A 1D spectrum instance containing the histogram.

See also:

[hyperspy.api.signals.BaseSignal.print_summary_statistics](#)
[numpy.histogram](#), [dask.array.histogram](#)

Examples

```
>>> s = hs.signals.Signal1D(np.random.normal(size=(10, 100)))
>>> # Plot the data histogram
>>> s.get_histogram().plot()
>>> # Plot the histogram of the signal at the current coordinates
>>> s.get_current_signal().get_histogram().plot()
```

`get_noise_variance()`

Get the noise variance of the signal, if set.

Equivalent to `s.metadata.Signal.Noise_properties.variance`.

Parameters

None

Returns

variance

[None or float or *BaseSignal* (or subclass)] Noise variance of the signal, if set. Otherwise returns None.

`ifft(shift=None, return_real=True, **kwargs)`

Compute the inverse discrete Fourier Transform.

This function computes the real part of the inverse of the discrete Fourier Transform over the signal axes by means of the Fast Fourier Transform (FFT) as implemented in numpy.

Parameters

shift

[bool or None, optional] If None, the shift option will be set to the original status of the FFT using the value in metadata. If no FFT entry is present in metadata, the parameter will be set to False. If True, the origin of the FFT will be shifted to the centre. If False, the origin will be kept at (0, 0) (default is None).

return_real

[bool, default True] If True, returns only the real part of the inverse FFT. If False, returns all parts.

**kwargs

[dict] other keyword arguments are described in `numpy.fft.ifftn()`

Returns

s

[*BaseSignal* (or subclass)] A Signal containing the result of the inverse FFT algorithm

Raises

`NotImplementedError`

If performing IFFT along a non-uniform axis.

Notes

Requires a uniform axis. For further information see the documentation of `numpy.fft.ifftn()`

Examples

```
>>> import skimage
>>> im = hs.signals.Signal2D(skimage.data.camera())
>>> imfft = im.fft()
>>> imfft.ifft()
<Signal2D, title: real(iFFT of FFT of ), dimensions: (|512, 512)>
```

indexmax(*axis*, *out=None*, *rechunk=False*)

Returns a signal with the index of the maximum along an axis.

Parameters

axis

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

s

[*BaseSignal* (or subclass)] A new Signal containing the indices of the maximum along the specified axis. Note: the data *dtype* is always *int*.

See also:

max, *min*, *sum*, *mean*, *std*, *var*, *indexmin*, *valuemax*, *valuemin*

Examples

```
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.indexmax(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

indexmin(*axis*, *out=None*, *rechunk=False*)

Returns a signal with the index of the minimum along an axis.

Parameters

axis

[[int](#), [str](#), or [DataAxis](#)] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

Returns

s

[[BaseSignal](#) (or subclass)] A new Signal containing the indices of the minimum along the specified axis. Note: the data *dtype* is always [int](#).

See also:

[max](#), [min](#), [sum](#), [mean](#), [std](#), [var](#), [indexmax](#), [valuemax](#), [valuemin](#)

Examples

```
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.indexmin(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

integrate1D(*axis*, *out=None*, *rechunk=False*)

Integrate the signal over the given axis.

The integration is performed using [Simpson's rule](#) if *axis.is_binned* is [False](#) and simple summation over the given axis if [True](#) (along binned axes, the detector already provides integrated counts per bin).

Parameters**axis**

[[int](#), [str](#), or [DataAxis](#)] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

Returns

s

[[BaseSignal](#) (or subclass)] A new Signal containing the integral of the provided Signal along the specified axis.

See also:

[*integrate_simpson, derivative*](#)

Examples

```
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.integrate1D(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

integrate_simpson(*axis*, *out=None*, *rechunk=False*)

Calculate the integral of a Signal along an axis using Simpson's rule.

Parameters

axis

[[int](#), [str](#), or [DataAxis](#)] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

Returns

s

[[BaseSignal](#) (or subclass)] A new Signal containing the integral of the provided Signal along the specified axis.

See also:

[*hyperspy.api.signals.BaseSignal.derivative*](#)
[*hyperspy.api.signals.BaseSignal.integrate1D*](#)

Examples

```
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.integrate_simpson(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

interpolate_on_axis(*new_axis*, *axis=0*, *inplace=False*, *degree=1*)

Replaces the given *axis* with the provided *new_axis* and interpolates data accordingly using [scipy.interpolate.make_interp_spline\(\)](#).

Parameters

new_axis

[*hyperspy.axes.UniformDataAxis*,]

:class:`hyperspy.axes.DataAxis` or :class:`hyperspy.axes.FunctionalDataAxis`

Axis which replaces the one specified by the `axis` argument. If this new axis exceeds the range of the old axis, a warning is raised that the data will be extrapolated.

axis

[`int` or `str`, default 0] Specifies the axis which will be replaced using the index of the axis in the *axes_manager*. The axis can be specified using the index of the axis in *axes_manager* or the axis name.

inplace

[`bool`, default `False`] If `True` the data of *self* is replaced by the result and the axis is changed inplace. Otherwise *self* is not changed and a new signal with the changes incorporated is returned.

degree: int, default 1

Specifies the B-Spline degree of the used interpolator.

Returns

***BaseSignal* (or subclass)**

A copy of the object with the axis exchanged and the data interpolated. This only occurs when `inplace` is set to `False`, otherwise nothing is returned.

property `is_rgb`

Whether or not this signal is an RGB *dtype*.

property `is_rgba`

Whether or not this signal is an RGB + alpha channel *dtype*.

property `is_rgbx`

Whether or not this signal is either an RGB or RGB + alpha channel *dtype*.

map(*function*, *show_progressbar*=None, *num_workers*=None, *inplace*=True, *ragged*=None, *navigation_chunks*=None, *output_signal_size*=None, *output_dtype*=None, *lazy_output*=None, ***kwargs*)

Apply a function to the signal data at all the navigation coordinates.

The function must operate on numpy arrays. It is applied to the data at each navigation coordinate pixel-pixel. Any extra keyword arguments are passed to the function. The keywords can take different values at different coordinates. If the function takes an *axis* or *axes* argument, the function is assumed to be vectorized and the signal axes are assigned to *axis* or *axes*. Otherwise, the signal is iterated over the navigation axes and a progress bar is displayed to monitor the progress.

In general, only navigation axes (order, calibration, and number) are guaranteed to be preserved.

Parameters

function

[`function`] Any function that can be applied to the signal. This function should not alter any mutable input arguments or input data. So do not do operations which alter the input, without copying it first. For example, instead of doing `image *= mask`, rather do `image = image * mask`. Likewise, do not do `image[5, 5] = 10` directly on the input data or arguments, but make a copy of it first. For example via `image = copy.deepcopy(image)`.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

lazy_output

[`None` or `bool`] If `True`, the output will be returned as a lazy signal. This means the calculation itself will be delayed until either `compute()` is used, or the signal is stored as a file. If `False`, the output will be returned as a non-lazy signal, this means the outputs will be calculated directly, and loaded into memory. If `None` the output will be lazy if the input signal is lazy, and non-lazy if the input signal is non-lazy.

inplace

[`bool`, default `True`] If `True`, the data is replaced by the result. Otherwise a new `Signal` with the results is returned.

ragged

[`None` or `bool`, default `None`] Indicates if the results for each navigation pixel are of identical shape (and/or numpy arrays to begin with). If `None`, the output signal will be ragged only if the original signal is ragged.

navigation_chunks

[`str`, `None`, `int` or `tuple` of `int`, default `None`] Set the `navigation_chunks` argument to a tuple of integers to split the navigation axes into chunks. This can be useful to enable using multiple cores with signals which are less than 100 MB. This argument is passed to `rechunk()`.

output_signal_size

[`None`, `tuple`] Since the size and dtype of the signal dimension of the output signal can be different from the input signal, this output signal size must be calculated somehow. If both `output_signal_size` and `output_dtype` is `None`, this is automatically determined. However, if for some reason this is not working correctly, this can be specified via `output_signal_size` and `output_dtype`. The most common reason for this failing is due to the signal size being different for different navigation positions. If this is the case, use `ragged=True`. `None` is default.

output_dtype

[`None`, `numpy.dtype`] See docstring for `output_signal_size` for more information. Default `None`.

num_workers

[`None` or `int`] Number of worker used by dask. If `None`, default to dask default value.

****kwargs**

[`dict`] All extra keyword arguments are passed to the provided function

Notes

If the function results do not have identical shapes, the result is an array of navigation shape, where each element corresponds to the result of the function (of arbitrary object type), called a “ragged array”. As such, most functions are not able to operate on the result and the data should be used directly.

This method is similar to Python’s `map()` that can also be utilized with a `BaseSignal` instance for similar purposes. However, this method has the advantage of being faster because it iterates the underlying numpy data array instead of the `BaseSignal`.

Currently requires a uniform axis.

Examples

Apply a Gaussian filter to all the images in the dataset. The sigma parameter is constant:

```
>>> import scipy.ndimage
>>> im = hs.signals.Signal2D(np.random.random((10, 64, 64)))
>>> im.map(scipy.ndimage.gaussian_filter, sigma=2.5)
```

Apply a Gaussian filter to all the images in the dataset. The signal parameter is variable:

```
>>> im = hs.signals.Signal2D(np.random.random((10, 64, 64)))
>>> sigmas = hs.signals.BaseSignal(np.linspace(2, 5, 10)).T
>>> im.map(scipy.ndimage.gaussian_filter, sigma=sigmas)
```

Rotate the two signal dimensions, with different amount as a function of navigation index. Delay the calculation by getting the output lazily. The calculation is then done using the compute method.

```
>>> from scipy.ndimage import rotate
>>> s = hs.signals.Signal2D(np.random.random((5, 4, 40, 40)))
>>> s_angle = hs.signals.BaseSignal(np.linspace(0, 90, 20).reshape(5, 4)).T
>>> s.map(rotate, angle=s_angle, reshape=False, lazy_output=True)
>>> s.compute()
```

Rotate the two signal dimensions, with different amount as a function of navigation index. In addition, the output is returned as a new signal, instead of replacing the old signal.

```
>>> s = hs.signals.Signal2D(np.random.random((5, 4, 40, 40)))
>>> s_angle = hs.signals.BaseSignal(np.linspace(0, 90, 20).reshape(5, 4)).T
>>> s_rot = s.map(rotate, angle=s_angle, reshape=False, inplace=False)
```

If you want some more control over computing a signal that isn't lazy you can always set lazy_output to True and then compute the signal setting the scheduler to 'threading', 'processes', 'single-threaded' or 'distributed'.

Additionally, you can set the navigation_chunks argument to a tuple of integers to split the navigation axes into chunks. This can be useful if your signal is less than 100 mb but you still want to use multiple cores.

```
>>> s = hs.signals.Signal2D(np.random.random((5, 4, 40, 40)))
>>> s_angle = hs.signals.BaseSignal(np.linspace(0, 90, 20).reshape(5, 4)).T
>>> s.map(
...     rotate, angle=s_angle, reshape=False, lazy_output=True,
...     inplace=True, navigation_chunks=(2,2)
... )
>>> s.compute()
```

max(axis=None, out=None, rechunk=False)

Returns a signal with the maximum of the signal along at least one axis.

Parameters

axis

[int, str, DataAxis or tuple] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in axes_manager or the name of the axis. Any duplicates are removed. If None, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

s

[*BaseSignal* (or subclass)] A new *Signal* containing the maximum of the provided *Signal* over the specified axes

See also:

min, *sum*, *mean*, *std*, *var*, *indexmax*, *indexmin*, *valuemax*, *valuemin*

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.max(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

mean(*axis=None*, *out=None*, *rechunk=False*)

Returns a signal with the average of the signal along at least one axis.

Parameters**axis**

[*int*, *str*, *DataAxis* or *tuple*] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If *None*, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

s

[*BaseSignal* (or subclass)] A new *Signal* containing the mean of the provided *Signal* over the specified axes

See also:

max, min, sum, std, var, indexmax, indexmin, valuemax, valuemin

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.mean(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

property metadata

The metadata of the signal.

min(*axis=None, out=None, rechunk=False*)

Returns a signal with the minimum of the signal along at least one axis.

Parameters

axis

[*int, str, DataAxis* or *tuple*] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If *None*, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

s

[*BaseSignal* (or subclass)] A new *Signal* containing the minimum of the provided *Signal* over the specified axes

See also:

max, sum, mean, std, var, indexmax, indexmin, valuemax, valuemin

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.min(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

nanmax(*axis=None, out=None, rechunk=False*)

Identical to `max()`, except ignores missing (NaN) values. See that method’s documentation for details.

nanmean(*axis=None, out=None, rechunk=False*)

Identical to `mean()`, except ignores missing (NaN) values. See that method’s documentation for details.

nanmin(*axis=None, out=None, rechunk=False*)

Identical to `min()`, except ignores missing (NaN) values. See that method’s documentation for details.

nanstd(*axis=None, out=None, rechunk=False*)

Identical to `std()`, except ignores missing (NaN) values. See that method’s documentation for details.

nansum(*axis=None, out=None, rechunk=False*)

Identical to `sum()`, except ignores missing (NaN) values. See that method’s documentation for details.

nanvar(*axis=None, out=None, rechunk=False*)

Identical to `var()`, except ignores missing (NaN) values. See that method’s documentation for details.

normalize_bss_components(*target='factors', function=<function sum>*)

Normalize BSS components.

Parameters

target

[{"factors", "loadings"}] Normalize components based on the scale of either the factors or loadings.

function

[`numpy callable()`, default `numpy.sum`] Each target component is divided by the output of `function(target)`. The function must return a scalar when operating on numpy arrays and must have an `axis` argument.

normalize_decomposition_components(*target='factors', function=<function sum>*)

Normalize decomposition components.

Parameters

target

[{"factors", "loadings"}] Normalize components based on the scale of either the factors or loadings.

function

[`numpy callable()`, default `numpy.sum`] Each target component is divided by the output of `function(target)`. The function must return a scalar when operating on numpy arrays and must have an `axis` argument.

normalize_poissonian_noise(*navigation_mask=None, signal_mask=None*)

Normalize the signal under the assumption of Poisson noise.

Scales the signal using to “normalize” the Poisson data for subsequent decomposition analysis [*].

Parameters

navigation_mask

[{`None`, `bool numpy array`}, default `None`] Optional mask applied in the navigation axis.

signal_mask

[{`None`, `bool numpy array`}, default `None`] Optional mask applied in the signal axis.

References

property `original_metadata`

The original metadata of the signal.

`plot(navigator='auto', axes_manager=None, plot_markers=True, **kwargs)`

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with `signal_dimension` equal to 0, 1 and 2 can be plotted.

Parameters

`navigator`

[`str`, `None`, or `BaseSignal` (or subclass).]

Allowed string values are `'auto'`, `'slider'`, and `'spectrum'`.

- If `'auto'`:
 - If `navigation_dimension > 0`, a navigator is provided to explore the data.
 - If `navigation_dimension` is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If `navigation_dimension` is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If `navigation_dimension` is > 1 , the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if `navigation_dimension > 2`, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”, and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.
 - For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If `'slider'`:
 - If `navigation_dimension > 0` a window with one slider per axis is raised to navigate the data.
- If `'spectrum'`:
 - If `navigation_dimension > 0` the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the `'auto'` option will be used instead.
- If `None`, no navigator will be provided.

Alternatively a `BaseSignal` (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the `navigation_shape + signal_shape` must be equal to the `navigator_shape` of the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to `'slider'`.

axes_manager

[None or *AxesManager*] If None, the signal's axes_manager attribute is used.

plot_markers

[bool, default True] Plot markers added using *s.add_marker(marker, permanent=True)*. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[dict] Only for image navigator, additional keyword arguments for *matplotlib.pyplot.imshow()*.

norm

[str, default 'auto'] The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. If 'auto', intensity is plotted on a linear scale except when *power_spectrum=True* (only for complex signals).

autoscale

[str] The string must contain any combination of the 'x' and 'v' characters. If 'x' or 'v' (for values) are in the string, the corresponding horizontal or vertical axis limits are set to their maxima and the axis limits will reset when the data or the navigation indices are changed. Default is 'v'.

****kwargs**

[dict] Only when plotting an image: additional (optional) keyword arguments for *matplotlib.pyplot.imshow()*.

plot_bss_factors (*comp_ids=None, calibrate=True, same_window=True, title=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, per_row=3, **kwargs*)

Plot factors from blind source separation results. In case of 1D signal axis, each factors line can be toggled on and off by clicking on their corresponding line in the legend.

Parameters**comp_ids**

[None, int, or list of int] If *comp_ids* is None, maps of all components will be returned. If it is an int, maps of components with ids from 0 to the given value will be returned. If *comp_ids* is a list of ints, maps of components with ids contained in the list will be returned.

calibrate

[bool] If True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

same_window

[bool] if True, plots each factor to the same window. They are not scaled. Default is True.

title

[str] Title of the matplotlib plot or label of the line in the legend when the dimension of factors is 1 and *same_window* is True.

cmap

[Colormap] The colormap used for the factor images, or for peak characteristics. Default is the matplotlib gray colormap (*plt.cm.gray*).

per_row

[int] The number of plots in each row, when the *same_window* parameter is True.

See also:

[plot_bss_loadings](#), [plot_bss_results](#)

plot_bss_loadings(*comp_ids=None, calibrate=True, same_window=True, title=None, with_factors=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', **kwargs*)

Plot loadings from blind source separation results. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on their corresponding line in the legend.

Parameters

comp_ids

[`None`, `int` or `list` of `int`] If `comp_ids=None`, maps of all components will be returned. If it is an `int`, maps of components with ids from 0 to the given value will be returned. If `comp_ids` is a list of `ints`, maps of components with ids contained in the list will be returned.

calibrate

[`bool`] if `True`, calibrates plots where calibration is available from the `axes_manager`. If `False`, plots are in pixels/channels.

same_window

[`bool`] If `True`, plots each factor to the same window. They are not scaled. Default is `True`.

title

[`str`] Title of the matplotlib plot or label of the line in the legend when the dimension of loadings is 1 and `same_window` is `True`.

with_factors

[`bool`] If `True`, also returns figure(s) with the factors for the given `comp_ids`.

cmap

[`Colormap`] The colormap used for the loading image, or for peak characteristics,. Default is the matplotlib gray colormap (`plt.cm.gray`).

no_nans

[`bool`] If `True`, removes NaN's from the loading plots.

per_row

[`int`] The number of plots in each row, when the `same_window` parameter is `True`.

axes_decor

[`str` or `None`, optional] One of: `'all'`, `'ticks'`, `'off'`, or `None` Controls how the axes are displayed on each image; default is `'all'` If `'all'`, both ticks and axis labels will be shown If `'ticks'`, no axis labels will be shown, but ticks/labels will If `'off'`, all decorations and frame will be disabled If `None`, no axis decorations will be shown, but ticks/frame will

See also:

[*plot_bss_factors*](#), [*plot_bss_results*](#)

plot_bss_results(*factors_navigator='smart_auto', loadings_navigator='smart_auto', factors_dim=2, loadings_dim=2*)

Plot the blind source separation factors and loadings.

Unlike [*plot_bss_factors\(\)*](#) and [*plot_bss_loadings\(\)*](#), this method displays one component at a time. Therefore it provides a more compact visualization than then other two methods. The loadings and factors are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

Parameters

factors_navigator

[*str*, *None*, or *BaseSignal* (or subclass)] One of: 'smart_auto', 'auto', *None*, 'spectrum' or a *BaseSignal* object. 'smart_auto' (default) displays sliders if the navigation dimension is less than 3. For a description of the other options see the *plot()* documentation for details.

loadings_navigator

[*str*, *None*, or *BaseSignal* (or subclass)] See the *factors_navigator* parameter

factors_dim

[*int*] Currently HyperSpy cannot plot a signal when the signal dimension is higher than two. Therefore, to visualize the BSS results when the factors or the loadings have signal dimension greater than 2, the data can be viewed as spectra (or images) by setting this parameter to 1 (or 2). (The default is 2)

loadings_dim

[*int*] See the *factors_dim* parameter

See also:

plot_bss_factors, *plot_bss_loadings*, *plot_decomposition_results*

plot_cluster_distances(*cluster_ids=None*, *calibrate=True*, *same_window=True*, *with_centers=False*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *no_nans=False*, *per_row=3*, *axes_decor='all'*, *title=None*, ***kwargs*)

Plot the euclidian distances to the centroid of each cluster.

In case of 1D navigation axis, each line can be toggled on and off by clicking on the corresponding line in the legend.

Parameters**cluster_ids**

[*None*, *int*, or *list* of *int*] if *None* (default), returns maps of all components using the *number_of_cluster* was defined when executing *cluster*. Otherwise it raises a *ValueError*. if *int*, returns maps of cluster labels with ids from 0 to given *int*. if *list* of *ints*, returns maps of cluster labels with ids in given *list*.

calibrate

[*bool*] if *True*, calibrates plots where calibration is available from the *axes_manager*. If *False*, plots are in pixels/channels.

same_window

[*bool*] if *True*, plots each factor to the same window. They are not scaled. Default is *True*.

title

[*str*] Title of the *matplotlib* plot or label of the line in the legend when the dimension of distance is 1 and *same_window* is *True*.

with_centers

[*bool*] If *True*, also returns figure(s) with the cluster centers for the given *cluster_ids*.

cmap

[*matplotlib.colors.Colormap*] The colormap used for the factor image, or for peak characteristics, the colormap used for the scatter plot of some peak characteristic.

no_nans

[*bool*] If *True*, removes NaN's from the loading plots.

per_row

[int] the number of plots in each row, when the same_window parameter is True.

axes_decor

[None or str {'all', 'ticks', 'off'}, optional] Controls how the axes are displayed on each image; default is 'all' If 'all', both ticks and axis labels will be shown If 'ticks', no axis labels will be shown, but ticks/labels will If 'off', all decorations and frame will be disabled If None, no axis decorations will be shown, but ticks/frame will

See also:

[plot_cluster_signals](#), [plot_cluster_results](#), [plot_cluster_labels](#)

plot_cluster_labels(*cluster_ids=None, calibrate=True, same_window=True, with_centers=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', title=None, **kwargs*)

Plot cluster labels from a cluster analysis. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on the legended line.

Parameters**cluster_ids**

[None, int, or list of int] if None (default), returns maps of all components using the number_of_cluster was defined when executing cluster. Otherwise it raises a ValueError. if int, returns maps of cluster labels with ids from 0 to given int. if list of ints, returns maps of cluster labels with ids in given list.

calibrate

[bool] if True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

same_window

[bool] if True, plots each factor to the same window. They are not scaled. Default is True.

title

[str] Title of the matplotlib plot or label of the line in the legend when the dimension of labels is 1 and same_window is True.

with_centers

[bool] If True, also returns figure(s) with the cluster centers for the given cluster_ids.

cmap

[matplotlib.colors.Colormap] The colormap used for the factor image, or for peak characteristics, the colormap used for the scatter plot of some peak characteristic.

no_nans

[bool] If True, removes NaN's from the loading plots.

per_row

[int] the number of plots in each row, when the same_window parameter is True.

axes_decor

[None or str {'all', 'ticks', 'off'}, default 'all'] Controls how the axes are displayed on each image; default is 'all' If 'all', both ticks and axis labels will be shown If 'ticks', no axis labels will be shown, but ticks/labels will If 'off', all decorations and frame will be disabled If None, no axis decorations will be shown, but ticks/frame will

See also:

[*plot_cluster_signals*](#), [*plot_cluster_results*](#)

plot_cluster_metric()

Plot the cluster metrics calculated using the *estimate_number_of_clusters()* method

See also:

[*estimate_number_of_clusters*](#), [*cluster_analysis*](#), [*get_cluster_labels*](#)
[*get_cluster_signals*](#), [*plot_cluster_results*](#), [*plot_cluster_signals*](#)
[*plot_cluster_labels*](#)

plot_cluster_results(*centers_navigator='smart_auto'*, *labels_navigator='smart_auto'*, *centers_dim=2*,
labels_dim=2)

Plot the cluster labels and centers.

Unlike *plot_cluster_labels()* and *plot_cluster_signals()*, this method displays one component at a time. Therefore it provides a more compact visualization than then other two methods. The labels and centers are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

Parameters

centers_navigator, labels_navigator

[None, {"smart_auto" | "auto" | "spectrum"} or *BaseSignal*, default "smart_auto"] "smart_auto" displays sliders if the navigation dimension is less than 3. For a description of the other options see *plot* documentation for details.

labels_dim, centers_dims

[int, default 2] Currently HyperSpy cannot plot signals of dimension higher than two. Therefore, to visualize the clustering results when the centers or the labels have signal dimension greater than 2 we can view the data as spectra(images) by setting this parameter to 1(2)

See also:

[*plot_cluster_signals*](#), [*plot_cluster_labels*](#)

plot_cluster_signals(*signal='mean'*, *cluster_ids=None*, *calibrate=True*, *same_window=True*,
title=None, *per_row=3*)

Plot centers from a cluster analysis.

Parameters

signal

[{"mean", "sum", "centroid"}, optional] If "mean" or "sum" return the mean signal or sum respectively over each cluster. If "centroid", returns the signals closest to the centroid.

cluster_ids

[None, int, or list of int] If None, returns maps of all clusters. If int, returns maps of clusters with ids from 0 to given int. If list of ints, returns maps of clusters with ids in given list.

calibrate

[bool, default True] If True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

same_window

[bool, default True] If True, plots each center to the same window. They are not scaled.

title

[None or str, default None] Title of the matplotlib plot or label of the line in the legend when the dimension of loadings is 1 and same_window is True.

per_row

[int, default 3] The number of plots in each row, when the same_window parameter is True.

See also:

[*plot_cluster_labels*](#)

plot_cumulative_explained_variance_ratio(*n=50*)

Plot cumulative explained variance up to n principal components.

Parameters**n**

[int] Number of principal components to show.

Returns**ax**

[matplotlib.axes] Axes object containing the cumulative explained variance plot.

See also:

[*plot_explained_variance_ratio*](#)

plot_decomposition_factors(*comp_ids=None, calibrate=True, same_window=True, title=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, per_row=3, **kwargs*)

Plot factors from a decomposition. In case of 1D signal axis, each factors line can be toggled on and off by clicking on their corresponding line in the legend.

Parameters**comp_ids**

[None, int or list of int] If *comp_ids* is None, maps of all components will be returned if the *output_dimension* was defined when executing *decomposition()*. Otherwise it raises a *ValueError*. If *comp_ids* is an int, maps of components with ids from 0 to the given value will be returned. If *comp_ids* is a list of ints, maps of components with ids contained in the list will be returned.

calibrate

[bool] If True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

same_window

[bool] If True, plots each factor to the same window. They are not scaled. Default is True.

title

[str] Title of the matplotlib plot or label of the line in the legend when the dimension of factors is 1 and same_window is True.

cmap

[Colormap] The colormap used for the factor images, or for peak characteristics. Default is the matplotlib gray colormap (`plt.cm.gray`).

per_row

[[int](#)] The number of plots in each row, when the *same_window* parameter is True.

See also:

[plot_decomposition_loadings](#), [plot_decomposition_results](#)

plot_decomposition_loadings(*comp_ids=None, calibrate=True, same_window=True, title=None, with_factors=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', **kwargs*)

Plot loadings from a decomposition. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on the legended line.

Parameters**comp_ids**

[[None](#), [int](#), or [list](#) of [int](#)] If *comp_ids* is [None](#), maps of all components will be returned if the *output_dimension* was defined when executing [decomposition\(\)](#). Otherwise it raises a [ValueError](#). If *comp_ids* is an [int](#), maps of components with ids from 0 to the given value will be returned. If *comp_ids* is a list of [ints](#), maps of components with ids contained in the list will be returned.

calibrate

[[bool](#)] if True, calibrates plots where calibration is available from the *axes_manager*. If False, plots are in pixels/channels.

same_window

[[bool](#)] if True, plots each factor to the same window. They are not scaled. Default is True.

title

[[str](#)] Title of the matplotlib plot or label of the line in the legend when the dimension of loadings is 1 and *same_window* is True.

with_factors

[[bool](#)] If True, also returns figure(s) with the factors for the given *comp_ids*.

cmap

[[Colormap](#)] The colormap used for the loadings images, or for peak characteristics. Default is the matplotlib gray colormap (`plt.cm.gray`).

no_nans

[[bool](#)] If True, removes NaN's from the loading plots.

per_row

[[int](#)] The number of plots in each row, when the *same_window* parameter is True.

axes_decor

[[str](#) or [None](#), optional] One of: 'all', 'ticks', 'off', or None Controls how the axes are displayed on each image; default is 'all' If 'all', both ticks and axis labels will be shown. If 'ticks', no axis labels will be shown, but ticks/labels will. If 'off', all decorations and frame will be disabled. If None, no axis decorations will be shown, but ticks/frame will.

See also:

[plot_decomposition_factors](#), [plot_decomposition_results](#)

```
plot_decomposition_results(factors_navigator='smart_auto', loadings_navigator='smart_auto',
                          factors_dim=2, loadings_dim=2)
```

Plot the decomposition factors and loadings.

Unlike `plot_decomposition_factors()` and `plot_decomposition_loadings()`, this method displays one component at a time. Therefore it provides a more compact visualization than the other two methods. The loadings and factors are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

Parameters

factors_navigator

[`str`, `None`, or `BaseSignal` (or subclass)] One of: 'smart_auto', 'auto', `None`, 'spectrum' or a `BaseSignal` object. 'smart_auto' (default) displays sliders if the navigation dimension is less than 3. For a description of the other options see the `plot()` documentation for details.

loadings_navigator

[`str`, `None`, or `BaseSignal` (or subclass)] See the `factors_navigator` parameter

factors_dim, loadings_dim

[`int`] Currently HyperSpy cannot plot a signal when the signal dimension is higher than two. Therefore, to visualize the BSS results when the factors or the loadings have signal dimension greater than 2, the data can be viewed as spectra (or images) by setting this parameter to 1 (or 2). (The default is 2)

See also:

[plot_decomposition_factors](#), [plot_decomposition_loadings](#), [plot_bss_results](#)

```
plot_explained_variance_ratio(n=30, log=True, threshold=0, hline='auto', vline=False,
                              xaxis_type='index', xaxis_labeling=None, signal_fmt=None,
                              noise_fmt=None, fig=None, ax=None, **kwargs)
```

Plot the decomposition explained variance ratio vs index number.

This is commonly known as a scree plot.

Read more in the [User Guide](#).

Parameters

n

[`int` or `None`] Number of components to plot. If `None`, all components will be plot

log

[`bool`, default `True`] If `True`, the y axis uses a log scale.

threshold

[`float` or `int`] Threshold used to determine how many components should be highlighted as signal (as opposed to noise). If a float (between 0 and 1), `threshold` will be interpreted as a cutoff value, defining the variance at which to draw a line showing the cutoff between signal and noise; the number of signal components will be automatically determined by the cutoff value. If an int, `threshold` is interpreted as the number of components to highlight as signal (and no cutoff line will be drawn)

hline: {'auto', True, False}

Whether or not to draw a horizontal line illustrating the variance cutoff for signal/noise determination. Default is to draw the line at the value given in `threshold` (if it is a float) and not draw in the case `threshold` is an int, or not given. If `True`, (and

threshold is an int), the line will be drawn through the last component defined as signal. If False, the line will not be drawn in any circumstance.

vline: bool, default False

Whether or not to draw a vertical line illustrating an estimate of the number of significant components. If True, the line will be drawn at the the knee or elbow position of the curve indicating the number of significant components. If False, the line will not be drawn in any circumstance.

xaxis_type

[{'index', 'number'}] Determines the type of labeling applied to the x-axis. If 'index', axis will be labeled starting at 0 (i.e. “pythonic index” labeling); if 'number', it will start at 1 (number labeling).

xaxis_labeling

[{'ordinal', 'cardinal', None}] Determines the format of the x-axis tick labels. If 'ordinal', “1st, 2nd, ...” will be used; if 'cardinal', “1, 2, ...” will be used. If None, an appropriate default will be selected.

signal_fmt

[dict] Dictionary of matplotlib formatting values for the signal components

noise_fmt

[dict] Dictionary of matplotlib formatting values for the noise components

fig

[matplotlib.figure.Figure or None] If None, a default figure will be created, otherwise will plot into fig

ax

[matplotlib.axes.Axes or None] If None, a default ax will be created, otherwise will plot into ax

****kwargs**

remaining keyword arguments are passed to `matplotlib.figure.Figure`

Returns

`matplotlib.axes.Axes`

Axes object containing the scree plot

See also:

[*decomposition*](#), [*get_explained_variance_ratio*](#), [*get_decomposition_loadings*](#), [*get_decomposition_factors*](#)

Examples

To generate a scree plot with customized symbols for signal vs. noise components and a modified cutoff threshold value:

```
>>> s = hs.load("some_spectrum_image")
>>> s.decomposition()
>>> s.plot_explained_variance_ratio(
...     n=40,
...     threshold=0.005,
...     signal_fmt={'marker': 'v', 's': 150, 'c': 'pink'},
```

(continues on next page)

(continued from previous page)

```
...     noise_fmt={'marker': '*', 's': 200, 'c': 'green'}
...     )
```

print_summary_statistics(*formatter='%.3g', rechunk=False*)

Prints the five-number summary statistics of the data, the mean, and the standard deviation.

Prints the mean, standard deviation (std), maximum (max), minimum (min), first quartile (Q1), median, and third quartile. nans are removed from the calculations.

Parameters

formatter

[[str](#)] The number formatter to use for the output

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

See also:

[get_histogram](#)

property ragged

Whether the signal is ragged or not.

rebin(*new_shape=None, scale=None, crop=True, dtype=None, out=None*)

Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify **either** `new_shape` or `scale`. Scale of 1 means no binning and scale less than one results in up-sampling.

Parameters

new_shape

[[list](#) (of [float](#) or [int](#)) or `None`] For each dimension specify the `new_shape`. This will internally be converted into a `scale` parameter.

scale

[[list](#) (of [float](#) or [int](#)) or `None`] For each dimension, specify the new:old pixel ratio, e.g. a ratio of 1 is no binning and a ratio of 2 means that each pixel in the new spectrum is twice the size of the pixels in the old spectrum. The length of the list should match the dimension of the Signal's underlying data array. *Note : Only one of ``scale`` or ``new_shape`` should be specified, otherwise the function will not run*

crop

[[bool](#)] Whether or not to crop the resulting rebinned data (default is `True`). When binning by a non-integer number of pixels it is likely that the final row in each dimension will contain fewer than the full quota to fill one pixel. For example, a 5*5 array binned by 2.1 will produce two rows containing 2.1 pixels and one row containing only 0.8 pixels. Selection of `crop=True` or `crop=False` determines whether or not this "black" line is cropped from the final binned array or not. *Please note that if ``crop=False`` is used, the final row in each dimension may appear black if a fractional number of pixels are left over. It can be removed but has been left to preserve total counts before and after binning.*

dtype

[`None`, [numpy.dtype](#), "same"] Specify the dtype of the output. If `None`, the dtype will be determined by the behaviour of `numpy.sum()`, if "same", the dtype will be kept the same. Default is `None`.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

Returns*BaseSignal*

The resulting cropped signal.

Raises*NotImplementedError*

If trying to rebin over a non-uniform axis.

Examples

```
>>> spectrum = hs.signals.Signal1D(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<Signal1D, title: , dimensions: (4, 4|10)>
>>> print('Sum =', sum(sum(sum(spectrum.data))))
Sum = 164.0
```

```
>>> scale = [2, 2, 5]
>>> test = spectrum.rebin(scale)
>>> print(test)
<Signal1D, title: , dimensions: (2, 2|5)>
>>> print('Sum =', sum(sum(sum(test.data))))
Sum = 164.0
```

```
>>> s = hs.signals.Signal1D(np.ones((2, 5, 10), dtype=np.uint8))
>>> print(s)
<Signal1D, title: , dimensions: (5, 2|10)>
>>> print(s.data.dtype)
uint8
```

Use `dtype=np.uint16` to specify a dtype

```
>>> s2 = s.rebin(scale=(5, 2, 1), dtype=np.uint16)
>>> print(s2.data.dtype)
uint16
```

Use `dtype="same"` to keep the same dtype

```
>>> s3 = s.rebin(scale=(5, 2, 1), dtype="same")
>>> print(s3.data.dtype)
uint8
```

By default `dtype=None`, the dtype is determined by the behaviour of `numpy.sum`, in this case, unsigned integer of the same precision as the platform integer

```
>>> s4 = s.rebin(scale=(5, 2, 1))
>>> print(s4.data.dtype)
uint32
```

reverse_bss_component(*component_number*)

Reverse the independent component.

Parameters

component_number
[list or int] component index/es

Examples

```
>>> s = hs.load('some_file')
>>> s.decomposition(True)
>>> s.blind_source_separation(3)
```

Reverse component 1

```
>>> s.reverse_bss_component(1)
```

Reverse components 0 and 2

```
>>> s.reverse_bss_component((0, 2))
```

reverse_decomposition_component(*component_number*)

Reverse the decomposition component.

Parameters

component_number
[list or int] component index/es

Examples

```
>>> s = hs.load('some_file')
>>> s.decomposition(True)
```

Reverse component 1

```
>>> s.reverse_decomposition_component(1)
```

Reverse components 0 and 2

```
>>> s.reverse_decomposition_component((0, 2))
```

rollaxis(*axis*, *to_axis*, *optimize=False*)

Roll the specified axis backwards, until it lies in a given position.

Parameters

axis
[int, str, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name. The axis to roll backwards. The positions of the other axes do not change relative to one another.

to_axis
[int, str, or *DataAxis*] The axis can be passed directly, or specified using the index

of the axis in the Signal's *axes_manager* or the axis name. The axis is rolled until it lies before this other axis.

optimize

[bool] If True, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if True, the chunks are optimised for the new axes configuration.

Returns

s
[BaseSignal (or subclass)] Output signal.

See also:

[numpy.roll\(\)](#), [swap_axes](#)

Examples

```
>>> s = hs.signals.Signal1D(np.ones((5, 4, 3, 6)))
>>> s
<Signal1D, title: , dimensions: (3, 4, 5|6)>
>>> s.rollaxis(3, 1)
<Signal1D, title: , dimensions: (3, 4, 5|6)>
>>> s.rollaxis(2, 0)
<Signal1D, title: , dimensions: (5, 3, 4|6)>
```

save(filename=None, overwrite=None, extension=None, file_format=None, **kws)

Saves the signal in the specified format.

The function gets the format from the specified extension (see [Supported formats](#) in the User Guide for more information):

- 'hspy' for HyperSpy's HDF5 specification
- 'rpl' for Ripple (useful to export to Digital Micrograph)
- 'msa' for EMSA/MSA single spectrum saving.
- 'unf' for SEMPER unf binary format.
- 'blo' for Blockfile diffraction stack saving.
- Many image formats such as 'png', 'tiff', 'jpeg'...

If no extension is provided the default file format as defined in the *preferences* is used. Please note that not all the formats supports saving datasets of arbitrary dimensions, e.g. 'msa' only supports 1D data, and blockfiles only supports image stacks with a *navigation_dimension* < 2.

Each format accepts a different set of parameters. For details see the specific format documentation.

Parameters

filename

[str or None] If None (default) and *tmp_parameters.filename* and *tmp_parameters.folder* are defined, the filename and path will be taken from

there. A valid extension can be provided e.g. 'my_file.rpl' (see *extension* parameter).

overwrite

[*None* or *bool*] If *None*, if the file exists it will query the user. If *True*(*False*) it does(not) overwrite the file if it exists.

extension

[*None* or *str*] The extension of the file that defines the file format. Allowable string values are: {'hspy', 'hdf5', 'rpl', 'msa', 'unf', 'blo', 'emd', and common image extensions e.g. 'tiff', 'png', etc.} 'hspy' and 'hdf5' are equivalent. Use 'hdf5' if compatibility with HyperSpy versions older than 1.2 is required. If *None*, the extension is determined from the following list in this order:

- i) the filename
- ii) *Signal.tmp_parameters.extension*
- iii) 'hspy' (the default extension)

chunks

[*tuple* or *True* or *None* (default)] HyperSpy, Nexus and EMD NCEM format only. Define chunks used when saving. The chunk shape should follow the order of the array (*s.data.shape*), not the shape of the *axes_manager*. If *None* and lazy signal, the dask array chunking is used. If *None* and non-lazy signal, the chunks are estimated automatically to have at least one chunk per signal space. If *True*, the chunking is determined by the the *h5py guess_chunk* function.

save_original_metadata

[*bool*, default][*False*] Nexus file only. Option to save *hyperspy.original_metadata* with the signal. A loaded Nexus file may have a large amount of data when loaded which you may wish to omit on saving

use_default

[*bool*, default][*False*] Nexus file only. Define the default dataset in the file. If set to *True* the signal or first signal in the list of signals will be defined as the default (following Nexus v3 data rules).

write_dataset

[*bool*, optional] Only for hspy files. If *True*, write the dataset, otherwise, don't write it. Useful to save attributes without having to write the whole dataset. Default is *True*.

close_file

[*bool*, optional] Only for hdf5-based files and some zarr store. Close the file after writing. Default is *True*.

file_format: string

The file format of choice to save the file. If not given, it is inferred from the file extension.

set_noise_variance(*variance*)

Set the noise variance of the signal.

Equivalent to `s.metadata.set_item("Signal.Noise_properties.variance", variance)`.

Parameters**variance**

[*None* or *float* or *BaseSignal* (or subclass)] Value or values of the noise variance. A value of *None* is equivalent to clearing the variance.

Returns

None

set_signal_origin(*origin*)Set the *signal_origin* metadata value.The *signal_origin* attribute specifies if the data was obtained through experiment or simulation.**Parameters****origin**[*str*] Typically 'experiment' or 'simulation'**set_signal_type**(*signal_type*="")

Set the signal type and convert the current signal accordingly.

The *signal_type* attribute specifies the type of data that the signal contains e.g. electron energy-loss spectroscopy data, photoemission spectroscopy data, etc.When setting *signal_type* to a “known” type, HyperSpy converts the current signal to the most appropriate *BaseSignal* subclass. Known signal types are signal types that have a specialized *BaseSignal* subclass associated, usually providing specific features for the analysis of that type of signal.HyperSpy ships with a minimal set of known signal types. External packages can register extra signal types. To print a list of registered signal types in the current installation, call `print_known_signal_types()`, and see the developer guide for details on how to add new *signal_types*. A non-exhaustive list of HyperSpy extensions is also maintained here: <https://github.com/hyperspy/hyperspy-extensions-list>.**Parameters****signal_type**[*str*, optional] If no arguments are passed, the *signal_type* is set to undefined and the current signal converted to a generic signal subclass. Otherwise, set the *signal_type* to the given signal type or to the signal type corresponding to the given signal type alias. Setting the *signal_type* to a known signal type (if exists) is highly advisable. If none exists, it is good practice to set *signal_type* to a value that best describes the data signal type.**See also:**`hyperspy.api.print_known_signal_types`**Examples**

Let's first print all known signal types:

```
>>> s = hs.signals.Signal1D([0, 1, 2, 3])
>>> s
<Signal1D, title: , dimensions: (|4)>
>>> hs.print_known_signal_types()
+-----+-----+-----+-----+
| signal_type | aliases | class name | package |
+-----+-----+-----+-----+
| DielectricFunction | dielectric function | DielectricFunction | exspy |
| EDS_SEM | | EDSSEMSpectrum | exspy |
| EDS_TEM | | EDSTEMSpectrum | exspy |
| EELS | TEM EELS | EELSSpectrum | exspy |
| hologram | | HologramImage | holospy |
+-----+-----+-----+-----+
```

We can set the `signal_type` using the `signal_type`:

```
>>> s.set_signal_type("EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
>>> s.set_signal_type("EDS_SEM")
>>> s
<EDSSEMSpectrum, title: , dimensions: (|4)>
```

or any of its aliases:

```
>>> s.set_signal_type("TEM EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
```

To set the `signal_type` to “undefined”, simply call the method without arguments:

```
>>> s.set_signal_type()
>>> s
<Signal1D, title: , dimensions: (|4)>
```

split(*axis='auto', number_of_parts='auto', step_sizes='auto'*)

Splits the data into several signals.

The split can be defined by giving the *number_of_parts*, a homogeneous step size, or a list of customized step sizes. By default ('auto'), the function is the reverse of `stack()`.

Parameters

axis

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name. If 'auto' and if the object has been created with `stack()` (and `stack_metadata=True`), this method will return the former list of signals (information stored in *meta-data.HyperSpy.Stacking_history*). If it was not created with `stack()`, the last navigation axis will be used.

number_of_parts

[*str* or *int*] Number of parts in which the spectrum image will be split. The splitting is homogeneous. When the axis size is not divisible by the *number_of_parts* the remainder data is lost without warning. If *number_of_parts* and *step_sizes* is 'auto', *number_of_parts* equals the length of the axis, *step_sizes* equals one, and the axis is suppressed from each sub-spectrum.

step_sizes

[*str*, *list* (of *int*), or *int*] Size of the split parts. If 'auto', the *step_sizes* equals one. If an *int* is given, the splitting is homogeneous.

Returns

list of *BaseSignal*

A list of the split signals

Raises

NotImplementedError

If trying to split along a non-uniform axis.

Examples

```

>>> s = hs.signals.Signal1D(np.random.random([4, 3, 2]))
>>> s
<Signal1D, title: , dimensions: (3, 4|2)>
>>> s.split()
[<Signal1D, title: , dimensions: (3|2)>, <Signal1D, title: , dimensions: (3|2)>
 →, <Signal1D, title: , dimensions: (3|2)>, <Signal1D, title: , dimensions:
 →(3|2)>]
>>> s.split(step_sizes=2)
[<Signal1D, title: , dimensions: (3, 2|2)>, <Signal1D, title: , dimensions: (3,
 → 2|2)>]
>>> s.split(step_sizes=[1, 2])
[<Signal1D, title: , dimensions: (3, 1|2)>, <Signal1D, title: , dimensions: (3,
 → 2|2)>]

```

squeeze()

Remove single-dimensional entries from the shape of an array and the axes. See `numpy.squeeze()` for more details.

Returns

`s`
 [signal] A new signal object with single-entry dimensions removed

Examples

```

>>> s = hs.signals.Signal2D(np.random.random((2, 1, 1, 6, 8, 8)))
>>> s
<Signal2D, title: , dimensions: (6, 1, 1, 2|8, 8)>
>>> s = s.squeeze()
>>> s
<Signal2D, title: , dimensions: (6, 2|8, 8)>

```

std(*axis=None, out=None, rechunk=False*)

Returns a signal with the standard deviation of the signal along at least one axis.

Parameters

axis

[int, str, *DataAxis* or tuple] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If `None`, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or `None`] If `None`, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[bool] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

Returns

s

[*BaseSignal* (or subclass)] A new Signal containing the standard deviation of the provided Signal over the specified axes

See also:

max, min, sum, mean, var, indexmax, indexmin, valuemax, valuemin

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.std(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

sum(*axis=None, out=None, rechunk=False*)

Sum the data over the given axes.

Parameters

axis

[*int, str, DataAxis* or *tuple*] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If *None*, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

BaseSignal

A new Signal containing the sum of the provided Signal along the specified axes.

See also:

max, min, mean, std, var, indexmax, indexmin, valuemax, valuemin

Notes

If you intend to calculate the numerical integral of an unbinned signal, please use the `integrate1D()` function instead. To avoid erroneous misuse of the `sum` function as integral, it raises a warning when working with an unbinned, non-uniform axis.

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.sum(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

swap_axes(*axis1*, *axis2*, *optimize=False*)

Swap two axes in the signal.

Parameters

axis1: `:class:`int``, `:class:`str``, or `:class:`~hyperspy.axes.DataAxis``

The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

axis2: `:class:`int``, `:class:`str``, or `:class:`~hyperspy.axes.DataAxis``

The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

optimize

[`bool`] If `True`, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if `True`, the chunks are optimised for the new axes configuration.

Returns

`s`
[*BaseSignal* (or subclass)] A copy of the object with the axes swapped.

See also:

[*rollaxis*](#)

to_device()

Transfer data array from host to GPU device memory using `cupy.asarray`. Lazy signals are not supported by this method, see user guide for information on how to process data lazily using the GPU.

Returns

`None`.

Raises

BaseException

Raise exception if `cupy` is not installed.

BaseException

Raise exception if signal is lazy.

to_host()

Transfer data array from GPU device to host memory.

Returns

None.

Raises**BaseException**

Raise exception if signal is lazy.

transpose(*signal_axes=None, navigation_axes=None, optimize=False*)

Transposes the signal to have the required signal and navigation axes.

Parameters**signal_axes**

[None, int, or iterable type] The number (or indices) of axes to convert to signal axes

navigation_axes

[None, int, or iterable type] The number (or indices) of axes to convert to navigation axes

optimize

[bool] If True, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if True, the chunks are optimised for the new axes configuration.

See also:

T, as_signal2D, as_signal1D

Notes

With the exception of both axes parameters (*signal_axes* and *navigation_axes* getting iterables, generally one has to be None (i.e. “floating”). The other one specifies either the required number or explicitly the indices of axes to move to the corresponding space. If both are iterables, full control is given as long as all axes are assigned to one space only.

Examples

```
>>> # just create a signal with many distinct dimensions
>>> s = hs.signals.BaseSignal(np.random.rand(1,2,3,4,5,6,7,8,9))
>>> s
<BaseSignal, title: , dimensions: (|9, 8, 7, 6, 5, 4, 3, 2, 1)>
```

```
>>> s.transpose() # swap signal and navigation spaces
<BaseSignal, title: , dimensions: (9, 8, 7, 6, 5, 4, 3, 2, 1|)>
```

```
>>> s.T # a shortcut for no arguments
<BaseSignal, title: , dimensions: (9, 8, 7, 6, 5, 4, 3, 2, 1)>
```

```
>>> # roll to leave 5 axes in navigation space
>>> s.transpose(signal_axes=5)
<BaseSignal, title: , dimensions: (4, 3, 2, 1|9, 8, 7, 6, 5)>
```

```
>>> # roll leave 3 axes in navigation space
>>> s.transpose(navigation_axes=3)
<BaseSignal, title: , dimensions: (3, 2, 1|9, 8, 7, 6, 5, 4)>
```

```
>>> # 3 explicitly defined axes in signal space
>>> s.transpose(signal_axes=[0, 2, 6])
<BaseSignal, title: , dimensions: (8, 6, 5, 4, 2, 1|9, 7, 3)>
```

```
>>> # A mix of two lists, but specifying all axes explicitly
>>> # The order of axes is preserved in both lists
>>> s.transpose(navigation_axes=[1, 2, 3, 4, 5, 8], signal_axes=[0, 6, 7])
<BaseSignal, title: , dimensions: (8, 7, 6, 5, 4, 1|9, 3, 2)>
```

undo_treatments()

Undo Poisson noise normalization and other pre-treatments.

Only valid if calling `s.decomposition(..., copy=True)`.

unfold(unfold_navigation=True, unfold_signal=True)

Modifies the shape of the data by unfolding the signal and navigation dimensions separately

Parameters

unfold_navigation

[bool] Whether or not to unfold the navigation dimension(s) (default: True)

unfold_signal

[bool] Whether or not to unfold the signal dimension(s) (default: True)

Returns

needed_unfolding

[bool] Whether or not one of the axes needed unfolding (and that unfolding was performed)

Notes

It doesn't make sense to perform an unfolding when the total number of dimensions is < 2.

unfold_navigation_space()

Modify the shape of the data to obtain a navigation space of dimension 1

Returns

needed_unfolding

[bool] Whether or not the navigation space needed unfolding (and whether it was performed)

unfold_signal_space()

Modify the shape of the data to obtain a signal space of dimension 1

Returns**needed_unfolding**

[bool] Whether or not the signal space needed unfolding (and whether it was performed)

unfolded(*unfold_navigation=True, unfold_signal=True*)

Use this function together with a *with* statement to have the signal be unfolded for the scope of the *with* block, before automatically refolding when passing out of scope.

See also:

unfold, fold

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> with s.unfolded():
...     # Do whatever needs doing while unfolded here
...     pass
```

update_plot()

If this Signal has been plotted, update the signal and navigator plots, as appropriate.

valuemax(*axis, out=None, rechunk=False*)

Returns a signal with the value of coordinates of the maximum along an axis.

Parameters**axis**

[int, str, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[bool] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns**s**

[*BaseSignal* (or subclass)] A new Signal containing the calibrated coordinate values of the maximum along the specified axis.

See also:

```
hyperspy.api.signals.BaseSignal.max, hyperspy.api.signals.BaseSignal.min
hyperspy.api.signals.BaseSignal.sum, hyperspy.api.signals.BaseSignal.mean
hyperspy.api.signals.BaseSignal.std, hyperspy.api.signals.BaseSignal.var
hyperspy.api.signals.BaseSignal.indexmax,
hyperspy.api.signals.BaseSignal.indexmin
hyperspy.api.signals.BaseSignal.valuemin
```

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.valuemax(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

valuemin(*axis*, *out=None*, *rechunk=False*)

Returns a signal with the value of coordinates of the minimum along an axis.

Parameters

axis

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

BaseSignal or subclass

A new Signal containing the calibrated coordinate values of the minimum along the specified axis.

See also:

```
hyperspy.api.signals.BaseSignal.max, hyperspy.api.signals.BaseSignal.min
hyperspy.api.signals.BaseSignal.sum, hyperspy.api.signals.BaseSignal.mean
hyperspy.api.signals.BaseSignal.std, hyperspy.api.signals.BaseSignal.var
hyperspy.api.signals.BaseSignal.indexmax,
hyperspy.api.signals.BaseSignal.indexmin
hyperspy.api.signals.BaseSignal.valuemax
```

var(*axis=None*, *out=None*, *rechunk=False*)

Returns a signal with the variances of the signal along at least one axis.

Parameters

axis

[*int*, *str*, *DataAxis* or *tuple*] Either one on its own, or many axes in a tuple can

be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If `None`, the operation is performed over all navigation axes (default).

out

[*BaseSignal* (or subclass) or `None`] If `None`, a new *Signal* is created with the result of the operation and returned (default). If a *Signal* is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[`bool`] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

Returns

`s`

[*BaseSignal* (or subclass)] A new *Signal* containing the variance of the provided *Signal* over the specified axes

See also:

max, *min*, *sum*, *mean*, *std*, *indexmax*, *indexmin*, *valuemax*, *valuemin*

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.var(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

23.8.2 ComplexSignal

```
class hyperspy.api.signals.ComplexSignal(*args, **kwargs)
```

Bases: *BaseSignal*

General signal class for complex data.

Create a signal instance.

Parameters**data**

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will be stored in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[*dict*, optional] A dictionary containing a set of parameters that will be stored in the `original_metadata` attribute. It typically contains all the parameters that have been imported from the original data file.

ragged

[*bool* or *None*, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the `attributes` dictionary. If *None*, it does nothing. Default is *None*.

property amplitude

Get/set the amplitude of the data.

angle(*deg=False*)

Return the angle (also known as phase or argument). If the data is real, the angle is 0 for positive values and 2π for negative values.

Parameters**deg**

[*bool*, default *False*] Return angle in degrees if *True*, radians if *False*.

Returns*BaseSignal*

The counterclockwise angle from the positive real axis on the complex plane, with dtype as `numpy.float64`.

argand_diagram(*size=[256, 256]*, *range=None*)

Calculate and plot Argand diagram of complex signal.

Parameters**size**

[*list* of *int*, optional] Size of the Argand plot in pixels. Default is [256, 256].

range

[*None*, *numpy.ndarray*, default *None*] The position of the edges of the diagram with shape (2, 2) or (2,). All values outside of this range will be considered outliers and not tallied in the histogram. If *None* use the minimum and maximum values.

Returns*Signal2D*

The Argand diagram

Examples

```
>>> import holospy as holo
>>> hologram = holo.data.Fe_needle_hologram()
>>> ref = holo.data.Fe_needle_reference_hologram()
>>> w = hologram.reconstruct_phase(ref)
>>> w.argand_diagram(range=[-3, 3]).plot()
```

change_dtype(*dtype*)

Change the data type.

Parameters

dtype

[[str](#) or [numpy.dtype](#)] Typecode or data-type to which the array is cast. For complex signals only other complex dtypes are allowed. If real valued properties are required use `real`, `imag`, `amplitude` and `phase` instead.

property imag

Get/set the imaginary part of the data.

property phase

Get/set the phase of the data.

plot(*power_spectrum=False, representation='cartesian', same_axes=True, fft_shift=False, navigator='auto', axes_manager=None, norm='auto', **kwargs*)

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with `signal_dimension` equal to 0, 1 and 2 can be plotted.

Parameters**power_spectrum**

[[bool](#), default `False`.] If `True`, plot the power spectrum instead of the actual signal, if `False`, plot the real and imaginary parts of the complex signal.

representation

[{'cartesian' | 'polar'}] Determines if the real and imaginary part of the complex data is plotted ('cartesian', default), or if the amplitude and phase should be used ('polar').

same_axes

[[bool](#), default `True`] If `True` (default) plot the real and imaginary parts (or amplitude and phase) in the same figure if the signal is one-dimensional.

fft_shift

[[bool](#), default `False`] If `True`, shift the zero-frequency component. See `numpy.fft.fftfreq()` for more details.

navigator

[[str](#), `None`, or `BaseSignal` (or subclass).]

Allowed string values are `''auto''`, `''slider''`, and `''spectrum''`.

- If 'auto':
 - If `navigation_dimension > 0`, a navigator is provided to explore the data.
 - If `navigation_dimension` is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If `navigation_dimension` is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If `navigation_dimension` is `> 1`, the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if `navigation_dimension > 2`, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”,

and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.

- For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If 'slider':
 - If `navigation_dimension > 0` a window with one slider per axis is raised to navigate the data.
- If 'spectrum':
 - If `navigation_dimension > 0` the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the 'auto' option will be used instead.
- If None, no navigator will be provided.

Alternatively a `BaseSignal` (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the `navigation_shape + signal_shape` must be equal to the `navigator_shape` of the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

axes_manager

[None or `AxesManager`] If None, the signal’s `axes_manager` attribute is used.

plot_markers

[bool, default `True`] Plot markers added using `s.add_marker(marker, permanent=True)`. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[dict] Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

****kwargs**

[dict] Only when plotting an image: additional (optional) keyword arguments for `matplotlib.pyplot.imshow()`.

property real

Get/set the real part of the data.

unwrapped_phase(*wrap_around=False, seed=None, show_progressbar=None, num_workers=None*)

Return the unwrapped phase as an appropriate HyperSpy signal.

Parameters

wrap_around

[bool or iterable of bool, default `False`] When an element of the sequence is `True`, the unwrapping process will regard the edges along the corresponding axis of the image to be connected and use this connectivity to guide the phase unwrapping process. If only a single boolean is given, it will apply to all axes. Wrap around is not supported for 1D arrays.

seed

[`numpy.random.Generator`, int or None, default `None`] Pass to the `rng` argument of the `unwrap_phase()` function. Unwrapping 2D or 3D images uses random initialization. This sets the seed of the PRNG to achieve deterministic behavior.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

num_workers

[`None` or `int`] Number of worker used by dask. If `None`, default to dask default value.

Returns***BaseSignal*** (or subclass)

The unwrapped phase.

Notes

Uses the `unwrap_phase()` function from *skimage*. The algorithm is based on Miguel Arevallilo Herraiz, David R. Burton, Michael J. Lalor, and Munther A. Gdeisat, “Fast two-dimensional phase-unwrapping algorithm based on sorting by reliability following a noncontinuous path”, *Journal Applied Optics*, Vol. 41, No. 35, pp. 7437, 2002

23.8.3 ComplexSignal1D

```
class hyperspy.api.signals.ComplexSignal1D(*args, **kwargs)
```

Bases: *ComplexSignal*, *CommonSignal1D*

Signal class for complex 1-dimensional data.

Create a signal instance.

Parameters**data**

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the `ragged` value in the `attributes` dictionary. If `None`, it does nothing. Default is `None`.

23.8.4 ComplexSignal2D

`class hyperspy.api.signals.ComplexSignal2D(*args, **kw)`

Bases: *ComplexSignal*, *CommonSignal2D*

Signal class for complex 2-dimensional data.

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the metadata attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the attributes dictionary. If `None`, it does nothing. Default is `None`.

`add_phase_ramp(ramp_x, ramp_y, offset=0)`

Add a linear phase ramp to the wave.

Parameters

ramp_x: float

Slope of the ramp in x-direction.

ramp_y: float

Slope of the ramp in y-direction.

offset: float, optional

Offset of the ramp at the fulcrum.

Notes

The fulcrum of the linear ramp is at the origin and the slopes are given in units of the axis with the corresponding scale taken into account. Both are available via the `:attr:`~hyperspy.api.signals.BaseSignal.axes_manager``.

`plot(power_spectrum=False, fft_shift=False, navigator='auto', plot_markers=True, autoscale='v', norm='auto', vmin=None, vmax=None, gamma=1.0, linthresh=0.01, linscale=0.1, scalebar=True, scalebar_color='white', axes_ticks=None, axes_off=False, axes_manager=None, no_nans=False, colorbar=True, centre_colormap='auto', min_aspect=0.1, **kwargs)`

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with `signal_dimension` equal to 0, 1 and 2 can be plotted.

Parameters

power_spectrum

[`bool`, default `False`.] If `True`, plot the power spectrum instead of the actual signal, if `False`, plot the real and imaginary parts of the complex signal.

representation

[{'cartesian' | 'polar'}] Determines if the real and imaginary part of the complex data is plotted ('cartesian', default), or if the amplitude and phase should be used ('polar').

same_axes

[`bool`, default `True`] If `True` (default) plot the real and imaginary parts (or amplitude and phase) in the same figure if the signal is one-dimensional.

fft_shift

[`bool`, default `False`] If `True`, shift the zero-frequency component. See `numpy.fft.fftshift()` for more details.

navigator

[`str`, `None`, or `BaseSignal` (or subclass).]

Allowed string values are `'auto'`, `'slider'`, and `'spectrum'`.

- If `'auto'`:
 - If `navigation_dimension > 0`, a navigator is provided to explore the data.
 - If `navigation_dimension` is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If `navigation_dimension` is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If `navigation_dimension` is `> 1`, the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if `navigation_dimension > 2`, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”, and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.
 - For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If `'slider'`:
 - If `navigation_dimension > 0` a window with one slider per axis is raised to navigate the data.
- If `'spectrum'`:
 - If `navigation_dimension > 0` the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the `'auto'` option will be used instead.

- If None, no navigator will be provided.

Alternatively a [BaseSignal](#) (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the `navigation_shape + signal_shape` must be equal to the `navigator_shape` of the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

axes_manager

[None or [AxesManager](#)] If None, the signal's `axes_manager` attribute is used.

plot_markers

[bool, default True] Plot markers added using `s.add_marker(marker, permanent=True)`. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[dict] Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

colorbar

[bool, optional] If true, a colorbar is plotted for non-RGB images.

autoscale

[str, optional] The string must contain any combination of the 'x', 'y' and 'v' characters. If 'x' or 'y' are in the string, the corresponding axis limits are set to cover the full range of the data at a given position. If 'v' (for values) is in the string, the contrast of the image will be set automatically according to `vmin`` and ``vmax`` when the data or navigation indices change. Default is 'v'.

norm

[str {"auto" | "linear" | "power" | "log" | "symlog"} or [matplotlib.colors.Normalize](#)] Set the norm of the image to display. If "auto", a linear scale is used except if when `power_spectrum=True` in case of complex data type. "symlog" can be used to display negative value on a negative scale - read [matplotlib.colors.SymLogNorm](#) and the `linthresh` and `linscale` parameter for more details.

vmin, vmax

[{scalar, str}, optional] `vmin` and `vmax` are used to normalise the displayed data. It can be a float or a string. If string, it should be formatted as 'xth', where 'x' must be an float in the [0, 100] range. 'x' is used to compute the x-th percentile of the data. See `numpy.percentile()` for more information.

gamma

[float, optional] Parameter used in the power-law normalisation when the parameter `norm="power"`. Read [matplotlib.colors.PowerNorm](#) for more details. Default value is 1.0.

linthresh

[float, optional] When used with `norm="symlog"`, define the range within which the plot is linear (to avoid having the plot go to infinity around zero). Default value is 0.01.

linscale

[float, optional] This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of powers of base to use for each half of the linear range. See [matplotlib.colors.SymLogNorm](#) for more details. Default value is 0.1.

scalebar

[bool, optional] If True and the units and scale of the x and y axes are the same a scale

bar is plotted.

scalebar_color

[`str`, optional] A valid MPL color string; will be used as the scalebar color.

axes_ticks

[{`None`, `bool`}, optional] If True, plot the axes ticks. If None axes_ticks are only plotted when the scale bar is not plotted. If False the axes ticks are never plotted.

axes_off

[`bool`, default `False`]

no_nans

[`bool`, optional] If True, set nans to zero for plotting.

centre_colormap

[`bool` or "auto"] If True the centre of the color scheme is set to zero. This is specially useful when using diverging color schemes. If "auto" (default), diverging color schemes are automatically centred.

min_aspect

[`float`, optional] Set the minimum aspect ratio of the image and the figure. To keep the image in the aspect limit the pixels are made rectangular.

****kwargs**

[`dict`] Only when plotting an image: additional (optional) keyword arguments for `matplotlib.pyplot.imshow()`.

23.8.5 Signal1D

`class hyperspy.api.signals.Signal1D(*args, **kwargs)`

Bases: *BaseSignal*, *CommonSignal1D*

General 1D signal class.

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[`[dict/axes]`, optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the metadata attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the attributes dictionary. If None, it does nothing. Default is None.

align1D(*start=None, end=None, reference_indices=None, max_shift=None, interpolate=True, number_of_interpolation_points=5, interpolation_method='linear', crop=True, expand=False, fill_value=nan, also_align=None, mask=None, show_progressbar=None, iterpath='serpentine'*)

Estimate the shifts in the signal axis using cross-correlation and use the estimation to align the data in place. This method can only estimate the shift by comparing unidimensional features that should not change the position.

To decrease memory usage, time of computation and improve accuracy it is convenient to select the feature of interest setting the `start` and `end` keywords. By default interpolation is used to obtain subpixel precision.

Parameters

start, end

[`int`, `float` or `None`] The limits of the interval. If `int` they are taken as the axis index. If `float` they are taken as the axis value.

reference_indices

[`tuple` of `int` or `None`] Defines the coordinates of the spectrum that will be used as reference. If `None` the spectrum at the current coordinates is used for this purpose.

max_shift

[`int`] “Saturation limit” for the shift.

interpolate

[`bool`] If `True`, interpolation is used to provide sub-pixel accuracy.

number_of_interpolation_points

[`int`] Number of interpolation points. Warning: making this number too big can saturate the memory

interpolation_method

[`str` or `int`] Specifies the kind of interpolation as a string (`'linear'`, `'nearest'`, `'zero'`, `'slinear'`, `'quadratic'`, `'cubic'`) or as an integer specifying the order of the spline interpolator to use.

crop

[`bool`] If `True` automatically crop the signal axis at both ends if needed.

expand

[`bool`] If `True`, the data will be expanded to fit all data after alignment. Overrides `crop` argument.

fill_value

[`float`] If `crop` is `False` fill the data outside of the original interval with the given value where needed.

also_align

[`list` of `BaseSignal`, `None`] A list of `BaseSignal` instances that has exactly the same dimensions as this one and that will be aligned using the shift map estimated using the this signal.

mask

[`BaseSignal` or `bool`] It must have `signal_dimension = 0` and `navigation_shape` equal to the current signal. Where `mask` is `True` the shift is not computed and set to `nan`.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

Returns

`numpy.ndarray`

The result of the estimation.

Raises

SignalDimensionError

If the signal dimension is not 1.

See also:

`estimate_shift1D`

calibrate(*display=True, toolkit=None*)

Calibrate the spectral dimension using a gui. It displays a window where the new calibration can be set by:

- setting the values of offset, units and scale directly
- or selecting a range by dragging the mouse on the spectrum figure and setting the new values for the given range limits

Parameters

display

[`bool`] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[`str, iterable of str or None`] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Raises

SignalDimensionError

If the signal dimension is not 1.

NotImplementedError

If called with a non-uniform axes.

Notes

For this method to work the `output_dimension` must be 1.

create_model(*dictionary=None*)

Create a model for the current data.

Returns

model

[*Model1D* instance.]

crop_signal(*args, **kwargs)

Crop in place in the signal space.

Parameters

left_value, right_value

[`int, float or None`] If int the values are taken as indices. If float they are converted to indices using the spectral axis calibration. If `left_value` is None crops from the beginning of the axis. If `right_value` is None crops up to the end of the axis. If both

are None the interactive cropping interface is activated enabling cropping the spectrum using a span selector in the signal plot.

Raises

SignalDimensionError

If the signal dimension is not 1.

estimate_peak_width(*factor=0.5*, *window=None*, *return_interval=False*, *show_progressbar=None*, *num_workers=None*)

Estimate the width of the highest intensity of peak of the spectra at a given fraction of its maximum.

It can be used with asymmetric peaks. For accurate results any background must be previously subtracted. The estimation is performed by interpolation using cubic splines.

Parameters

factor

[*float*, default 0.5] Normalized height (in interval [0, 1]) at which to estimate the width. The default (0.5) estimates the FWHM.

window

[*None* or *float*] The size of the window centred at the peak maximum used to perform the estimation. The window size must be chosen with care: if it is narrower than the width of the peak at some positions or if it is so wide that it includes other more intense peaks this method cannot compute the width and a NaN is stored instead.

return_interval

[*bool*] If True, returns 2 extra signals with the positions of the desired height fraction at the left and right of the peak.

show_progressbar

[*None* or *bool*] If True, display a progress bar. If None, the default from the preferences settings is used.

num_workers

[*None* or *int*] Number of worker used by dask. If None, default to dask default value.

Returns

float or list of float

width or [width, left, right], depending on the value of *return_interval*.

estimate_shift1D(*start=None*, *end=None*, *reference_indices=None*, *max_shift=None*, *interpolate=True*, *number_of_interpolation_points=5*, *mask=None*, *show_progressbar=None*, *num_workers=None*)

Estimate the shifts in the current signal axis using cross-correlation. This method can only estimate the shift by comparing unidimensional features that should not change the position in the signal axis. To decrease the memory usage, the time of computation and the accuracy of the results it is convenient to select the feature of interest providing sensible values for *start* and *end*. By default interpolation is used to obtain subpixel precision.

Parameters

start, end

[*int*, *float* or *None*, default *None*] The limits of the interval. If *int* they are taken as the axis index. If *float* they are taken as the axis value.

reference_indices

[*tuple* of *int* or *None*, default *None*] Defines the coordinates of the spectrum that

will be used as reference. If None the spectrum at the current coordinates is used for this purpose.

max_shift

[`int`] “Saturation limit” for the shift.

interpolate

[`bool`, default `True`] If True, interpolation is used to provide sub-pixel accuracy.

number_of_interpolation_points

[`int`] Number of interpolation points. Warning: making this number too big can saturate the memory

mask

[*BaseSignal* of `bool`.] It must have `signal_dimension = 0` and `navigation_shape` equal to the current signal. Where mask is True the shift is not computed and set to nan.

show_progressbar

[`None` or `bool`] If True, display a progress bar. If `None`, the default from the preferences settings is used.

num_workers

[`None` or `int`] Number of worker used by dask. If `None`, default to dask default value.

Returns**`numpy.ndarray`**

An array with the result of the estimation in the axis units. Although the computation is performed in batches if the signal is lazy, the result is computed in memory because it depends on the current state of the axes that could change later on in the workflow.

Raises**`SignalDimensionError`**

If the signal dimension is not 1.

`NotImplementedError`

If the signal axis is a non-uniform axis.

`filter_butterworth`(*cutoff_frequency_ratio=None, type='low', order=2, display=True, toolkit=None*)

Butterworth filter in place.

Parameters**display**

[`bool`] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[`str`, iterable of `str` or `None`] If `None` (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Raises**`SignalDimensionError`**

If the signal dimension is not 1.

`NotImplementedError`

If the signal axis is a non-uniform axis.

find_peaks1D_ohaver(*xdim=None, slope_thresh=0, amp_thresh=None, subchannel=True, medfilt_radius=5, maxpeakn=30000, peakgroup=10, num_workers=None*)

Find positive peaks along a 1D Signal. It detects peaks by looking for downward zero-crossings in the first derivative that exceed `slope_thresh`.

`slope_thresh` and `amp_thresh`, control sensitivity: higher values will neglect broad peaks (slope) and smaller features (amp), respectively.

Parameters

slope_thresh

[float, default 0] 1st derivative threshold to count the peak; higher values will neglect broader features.

amp_thresh

[float, optional] Intensity threshold below which peaks are ignored; higher values will neglect smaller features; default is set to 10% of max(y).

medfilt_radius

[int, default 5] Median filter window to apply to smooth the data (see `scipy.signal.medfilt()`); if 0, no filter will be applied.

peakgroup

[int, default 10] Number of points around the “top part” of the peak that is taken to estimate the peak height. For spikes or very narrow peaks, set `peakgroup` to 1 or 2; for broad or noisy peaks, make `peakgroup` larger to reduce the effect of noise.

maxpeakn

[int, default 5000] Number of maximum detectable peaks.

subchannel

[bool, default True] Whether to use subchannel precision or not.

num_workers

[None or int] Number of worker used by dask. If None, default to dask default value.

Returns

numpy.ndarray

Structured array of shape (npeaks) containing fields: ‘position’, ‘width’, and ‘height’ for each peak.

Raises

SignalDimensionError

If the signal dimension is not 1.

gaussian_filter(*FWHM*)

Applies a Gaussian filter in the spectral dimension in place.

Parameters

FWHM

[float] The Full Width at Half Maximum of the gaussian in the spectral axis units

Raises

ValueError

If FWHM is equal or less than zero.

SignalDimensionError

If the signal dimension is not 1.

NotImplementedError

If the signal axis is a non-uniform axis.

hanning_taper(*side='both', channels=None, offset=0*)

Apply a hanning taper to the data in place.

Parameters**side**

[{ 'left' | 'right' | 'both' }] Specify which side to use.

channels

[None or int] The number of channels to taper. If None 5% of the total number of channels are tapered.

offset

[int]

Returns

int

Raises**SignalDimensionError**

If the signal dimension is not 1.

interpolate_in_between(*start, end, delta=3, show_progressbar=None, num_workers=None, **kwargs*)

Replace the data in a given range by interpolation. The operation is performed in place.

Parameters**start, end**

[int or float] The limits of the interval. If int, they are taken as the axis index. If float, they are taken as the axis value.

delta

[int or float] The windows around the (start, end) to use for interpolation. If int, they are taken as index steps. If float, they are taken in units of the axis value.

show_progressbar

[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

num_workers

[None or int] Number of worker used by dask. If None, default to dask default value.

****kwargs**

[dict] All extra keyword arguments are passed to `scipy.interpolate.interp1d`. See the function documentation for details.

Raises**SignalDimensionError**

If the signal dimension is not 1.

plot(*navigator='auto', plot_markers=True, autoscale='v', norm='auto', axes_manager=None, navigator_kwds={}, **kwargs*)

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

navigator

[`str`, `None`, or `BaseSignal` (or subclass).]

Allowed string values are `''auto''`, `''slider''`, and `''spectrum''`.

- If `'auto'`:
 - If `navigation_dimension > 0`, a navigator is provided to explore the data.
 - If `navigation_dimension` is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If `navigation_dimension` is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If `navigation_dimension` is > 1 , the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if `navigation_dimension > 2`, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”, and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.
 - For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If `'slider'`:
 - If `navigation_dimension > 0` a window with one slider per axis is raised to navigate the data.
- If `'spectrum'`:
 - If `navigation_dimension > 0` the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the `'auto'` option will be used instead.
- If `None`, no navigator will be provided.

Alternatively a `BaseSignal` (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the `navigation_shape + signal_shape` must be equal to the `navigator_shape` of the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to `'slider'`.

axes_manager

[`None` or `AxesManager`] If `None`, the signal’s `axes_manager` attribute is used.

plot_markers

[`bool`, default `True`] Plot markers added using `s.add_marker(marker, permanent=True)`. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[`dict`] Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

norm

[**str**, default 'auto'] The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. If 'auto', intensity is plotted on a linear scale except when `power_spectrum=True` (only for complex signals).

autoscale

[**str**] The string must contain any combination of the 'x' and 'v' characters. If 'x' or 'v' (for values) are in the string, the corresponding horizontal or vertical axis limits are set to their maxima and the axis limits will reset when the data or the navigation indices are changed. Default is 'v'.

remove_background(*signal_range='interactive', background_type='Power law', polynomial_order=2, fast=True, zero_fill=False, plot_remainder=True, show_progressbar=None, return_model=False, display=True, toolkit=None*)

Remove the background, either in place using a GUI or returned as a new spectrum using the command line. The fast option is not accurate for most background types - except Gaussian, Offset and Power law - but it is useful to estimate the initial fitting parameters before performing a full fit.

Parameters**signal_range**

[“interactive”, **tuple** of **int** or **float**, optional] If this argument is not specified, the signal range has to be selected using a GUI. And the original spectrum will be replaced. If tuple is given, a spectrum will be returned.

background_type

[**str**] The type of component which should be used to fit the background. Possible components: Doniach, Gaussian, Lorentzian, Offset, Polynomial, PowerLaw, Exponential, SkewNormal, SplitVoigt, Voigt. If Polynomial is used, the polynomial order can be specified

polynomial_order

[**int**, default 2] Specify the polynomial order if a Polynomial background is used.

fast

[**bool**] If True, perform an approximative estimation of the parameters. If False, the signal is fitted using non-linear least squares afterwards. This is slower compared to the estimation but often more accurate.

zero_fill

[**bool**] If True, all spectral channels lower than the lower bound of the fitting range will be set to zero (this is the default behavior of Gatan's DigitalMicrograph). Setting this value to False allows for inspection of the quality of background fit throughout the pre-fitting region.

plot_remainder

[**bool**] If True, add a (green) line previewing the remainder signal after background removal. This preview is obtained from a Fast calculation so the result may be different if a NLLS calculation is finally performed.

return_model

[**bool**] If True, the background model is returned. The χ^2 can be obtained from this model using `chisq()`.

show_progressbar

[**None** or **bool**] If True, display a progress bar. If None, the default from the preferences settings is used.

display

[**bool**] If True, display the user interface widgets. If False, return the widgets container

in a dictionary, usually for customisation or testing.

toolkit

[`str`, iterable of `str` or `None`] If `None` (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Returns

`None` or (`hyperspy.api.signals.BaseSignal`, `hyperspy.models.model1d.Model1D`)

If `signal_range` is not 'interactive', the signal with background subtracted is returned. If `return_model=True`, returns the background model, otherwise, the GUI widget dictionary is returned if `display=False` - see the display parameter documentation.

Raises

`SignalDimensionError`

If the signal dimension is not 1.

Examples

Using GUI, replaces spectrum s

```
>>> s = hs.signals.Signal1D(range(1000))
>>> s.remove_background()
```

Using command line, returns a `Signal1D`:

```
>>> s.remove_background(
...     signal_range=(400,450), background_type='PowerLaw'
... )
<Signal1D, title: , dimensions: (|1000)>
```

Using a full model to fit the background:

```
>>> s.remove_background(signal_range=(400,450), fast=False)
<Signal1D, title: , dimensions: (|1000)>
```

Returns background subtracted and the model:

```
>>> s.remove_background(
...     signal_range=(400,450), fast=False, return_model=True
... )
(<Signal1D, title: , dimensions: (|1000)>, <Model1D>)
```

`shift1D`(`shift_array`, `interpolation_method='linear'`, `crop=True`, `expand=False`, `fill_value=nan`, `show_progressbar=None`, `num_workers=None`)

Shift the data in place over the signal axis by the amount specified by an array.

Parameters

`shift_array`

[`BaseSignal` or `numpy.ndarray`] An array containing the shifting amount. It must have the same `axes_manager.navigation_shape` `axes_manager._navigation_shape_in_array` shape.

interpolation_method

[**str** or **int**] Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic') or as an integer specifying the order of the spline interpolator to use.

crop

[**bool**] If True automatically crop the signal axis at both ends if needed.

expand

[**bool**] If True, the data will be expanded to fit all data after alignment. Overrides *crop*.

fill_value

[**float**] If *crop* is False fill the data outside of the original interval with the given value where needed.

show_progressbar

[**None** or **bool**] If True, display a progress bar. If **None**, the default from the preferences settings is used.

num_workers

[**None** or **int**] Number of worker used by dask. If **None**, default to dask default value.

Raises**SignalDimensionError**

If the signal dimension is not 1.

NotImplementedError

If the signal axis is a non-uniform axis.

smooth_lowess(*smoothing_parameter=None, number_of_iterations=None, show_progressbar=None, num_workers=None, display=True, toolkit=None*)

Lowess data smoothing in place. If *smoothing_parameter* or *number_of_iterations* are **None** the method is run in interactive mode.

Parameters**smoothing_parameter: float or None**

Between 0 and 1. The fraction of the data used when estimating each y-value.

number_of_iterations: int or None

The number of residual-based reweightings to perform.

show_progressbar

[**None** or **bool**] If True, display a progress bar. If **None**, the default from the preferences settings is used.

num_workers

[**None** or **int**] Number of worker used by dask. If **None**, default to dask default value.

display

[**bool**] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[**str**, **iterable** of **str** or **None**] If **None** (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Raises**SignalDimensionError**

If the signal dimension is not 1.

smooth_savitzky_golay(*polynomial_order=None, window_length=None, differential_order=0, num_workers=None, display=True, toolkit=None*)

Apply a Savitzky-Golay filter to the data in place. If *polynomial_order* or *window_length* or *differential_order* are None the method is run in interactive mode.

Parameters

polynomial_order

[[int](#), optional] The order of the polynomial used to fit the samples. *polyorder* must be less than *window_length*.

window_length

[[int](#), optional] The length of the filter window (i.e. the number of coefficients). *window_length* must be a positive odd integer.

differential_order: int, optional

The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.

num_workers

[[None](#) or [int](#)] Number of worker used by dask. If None, default to dask default value.

display

[[bool](#)] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[[str](#), iterable of [str](#) or [None](#)] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Raises

NotImplementedError

If the signal axis is a non-uniform axis.

Notes

More information about the filter in `scipy.signal.savgol_filter`.

smooth_tv(*smoothing_parameter=None, show_progressbar=None, num_workers=None, display=True, toolkit=None*)

Total variation data smoothing in place.

Parameters

smoothing_parameter: float or None

Denoising weight relative to L2 minimization. If None the method is run in interactive mode.

show_progressbar

[[None](#) or [bool](#)] If True, display a progress bar. If None, the default from the preferences settings is used.

num_workers

[[None](#) or [int](#)] Number of worker used by dask. If None, default to dask default value.

display

[[bool](#)] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[[str](#), iterable of [str](#) or [None](#)] If [None](#) (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

Raises**SignalDimensionError**

If the signal dimension is not 1.

NotImplementedError

If the signal axis is a non-uniform axis.

spikes_diagnosis(*signal_mask=None, navigation_mask=None, **kwargs*)

Plots a histogram to help in choosing the threshold for spikes removal.

Parameters**signal_mask**

[[numpy.ndarray](#) of [bool](#)] Restricts the operation to the signal locations not marked as [True](#) (masked).

navigation_mask

[[numpy.ndarray](#) of [bool](#)] Restricts the operation to the navigation locations not marked as [True](#) (masked).

****kwargs**

[[dict](#)] Keyword arguments pass to [get_histogram\(\)](#)

See also:[*spikes_removal_tool*](#)

spikes_removal_tool(*signal_mask=None, navigation_mask=None, threshold='auto', interactive=True, display=True, toolkit=None, **kwargs*)

Graphical interface to remove spikes from EELS spectra or luminescence data. If non-interactive, it removes all spikes.

Parameters**signal_mask**

[[numpy.ndarray](#) of [bool](#)] Restricts the operation to the signal locations not marked as [True](#) (masked).

navigation_mask

[[numpy.ndarray](#) of [bool](#)] Restricts the operation to the navigation locations not marked as [True](#) (masked).

threshold

[['auto'](#) or [int](#)] if [int](#) set the threshold value use for the detecting the spikes. If ["auto"](#), determine the threshold value as being the first zero value in the histogram obtained from the [spikes_diagnosis\(\)](#) method.

interactive

[[bool](#)] If [True](#), remove the spikes using the graphical user interface. If [False](#), remove all the spikes automatically, which can introduce artefacts if used with signal containing peak-like features. However, this can be mitigated by using the [signal_mask](#) argument to mask the signal of interest.

display

[*bool*] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str*, iterable of *str* or *None*] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

****kwargs**

[*dict*] Keyword arguments pass to `SpikesRemoval`.

See also:

`spikes_diagnosis()`

23.8.6 Signal2D

`class hyperspy.api.signals.Signal2D(*args, **kwargs)`

Bases: `BaseSignal`, `CommonSignal2D`

General 2D signal class.

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the `AxesManager` class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will be stored in the metadata attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will be stored in the original_metadata attribute. It typically contains all the parameters that have been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the attributes dictionary. If None, it does nothing. Default is None.

`add_ramp(ramp_x, ramp_y, offset=0)`

Add a linear ramp to the signal.

Parameters

ramp_x: float

Slope of the ramp in x-direction.

ramp_y: float

Slope of the ramp in y-direction.

offset: float, optional

Offset of the ramp at the signal fulcrum.

Notes

The fulcrum of the linear ramp is at the origin and the slopes are given in units of the axis with the according scale taken into account. Both are available via the *axes_manager* of the signal.

`align2D(crop=True, fill_value=nan, shifts=None, expand=False, interpolation_order=1, show_progressbar=None, num_workers=None, **kwargs)`

Align the images in-place using `scipy.ndimage.shift()`.

The images can be aligned using either user-provided shifts or by first estimating the shifts.

See `estimate_shift2D()` for more details on estimating image shifts.

Parameters**crop**

[bool] If True, the data will be cropped not to include regions with missing data

fill_value

[int, float, numpy.nan] The areas with missing data are filled with the given value. Default is np.nan.

shifts

[None or numpy.ndarray] The array of shifts must be in pixel units. The shape must be the navigation shape using numpy order convention. If None the shifts are estimated using `estimate_shift2D()`.

expand

[bool] If True, the data will be expanded to fit all data after alignment. Overrides `crop`.

interpolation_order: int

The order of the spline interpolation. Default is 1, linear interpolation.

show_progressbar

[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

num_workers

[None or int] Number of worker used by dask. If None, default to dask default value.

****kwargs**

[dict] Keyword arguments passed to `estimate_shift2D()`.

Returns**numpy.ndarray**

The estimated shifts are returned only if `shifts` is None

Raises**NotImplementedError**

If one of the signal axes is a non-uniform axis.

See also:

`estimate_shift2D`

calibrate(*x0=None, y0=None, x1=None, y1=None, new_length=None, units=None, interactive=True, display=True, toolkit=None*)

Calibrate the x and y signal dimensions.

Can be used either interactively, or by passing values as parameters.

Parameters

x0, y0, x1, y1

[`float`, `int`, optional] If `interactive` is `False`, these must be set. If given in floats the input will be in scaled axis values. If given in integers, the input will be in non-scaled pixel values. Similar to how integer and float input works when slicing using `isig` and `inav`.

new_length

[`scalar`, optional] If `interactive` is `False`, this must be set.

units

[`str`, optional] If `interactive` is `False`, this is used to set the axes units.

interactive

[`bool`, default `True`] If `True`, will use a plot with an interactive line for calibration. If `False`, `x0`, `y0`, `x1`, `y1` and `new_length` must be set.

display

[`bool`, default `True`]

toolkit

[`str`, optional]

Examples

```
>>> s = hs.signals.Signal2D(np.random.random((100, 100)))
>>> s.calibrate()
```

Running non-interactively

```
>>> s = hs.signals.Signal2D(np.random.random((100, 100)))
>>> s.calibrate(x0=10, y0=10, x1=60, y1=10, new_length=100,
...             interactive=False, units="nm")
```

create_model(*dictionary=None*)

Create a model for the current signal

Parameters

dictionary

[`None` or `dict`, optional] A dictionary to be used to recreate a model. Usually generated using `hyperspy.model.BaseModel.as_dictionary()`

Returns

`hyperspy.models.model2d.Model2D`

crop_signal(*top=None, bottom=None, left=None, right=None, convert_units=False*)

Crops in signal space and in place.

Parameters

top, bottom, left, right

[[int](#) or [float](#)] If int the values are taken as indices. If float the values are converted to indices.

convert_units

[[bool](#)] Default is False If True, convert the signal units using the 'convert_to_units' method of the *axes_manager*. If False, does nothing.

See also:

[*hyperspy.api.signals.BaseSignal.crop*](#)

estimate_shift2D(*reference='current', correlation_threshold=None, chunk_size=30, roi=None, normalize_corr=False, sobel=True, medfilter=True, hanning=True, plot=False, dtype='float', show_progressbar=None, sub_pixel_factor=1*)

Estimate the shifts in an image using phase correlation.

This method can only estimate the shift by comparing bi-dimensional features that should not change position between frames. To decrease the memory usage, the time of computation and the accuracy of the results it is convenient to select a region of interest by setting the *roi* argument.

Parameters**reference**

[{'current', 'cascade', 'stat'}] If 'current' (default) the image at the current coordinates is taken as reference. If 'cascade' each image is aligned with the previous one. If 'stat' the translation of every image with all the rest is estimated and by performing statistical analysis on the result the translation is estimated.

correlation_threshold

[[None](#), [str](#) or [float](#)] This parameter is only relevant when *reference='stat'*. If float, the shift estimations with a maximum correlation value lower than the given value are not used to compute the estimated shifts. If 'auto' the threshold is calculated automatically as the minimum maximum correlation value of the automatically selected reference image.

chunk_size

[[None](#) or [int](#)] If int and *reference='stat'* the number of images used as reference are limited to the given value.

roi

[[tuple](#) of [int](#) or [float](#)] Define the region of interest (left, right, top, bottom). If int (float), the position is given by axis index (value). Note that ROIs can be used in place of a tuple.

normalize_corr

[[bool](#), default [False](#)] If True, use phase correlation to align the images, otherwise use cross correlation.

sobel

[[bool](#), default [True](#)] Apply a Sobel filter for edge enhancement

medfilter

[[bool](#), default [True](#)] Apply a median filter for noise reduction

hanning

[[bool](#), default [True](#)] Apply a 2D hanning filter

plot

[`bool` or `str`] If True plots the images after applying the filters and the phase correlation. If 'reuse', it will also plot the images, but it will only use one figure, and continuously update the images in that figure as it progresses through the stack.

dtype

[`str` or `numpy.dtype`] Typecode or data-type in which the calculations must be performed.

show_progressbar

[`None` or `bool`] If True, display a progress bar. If None, the default from the preferences settings is used.

sub_pixel_factor

[`float`] Estimate shifts with a sub-pixel accuracy of $1/\text{sub_pixel_factor}$ parts of a pixel. Default is 1, i.e. no sub-pixel accuracy.

Returns`numpy.ndarray`

Estimated shifts in pixels.

See also:

[`align2D`](#)

Notes

The statistical analysis approach to the translation estimation when using `reference='stat'` roughly follows⁰. If you use it please cite their article.

References

find_peaks(*method='local_max', interactive=True, current_index=False, show_progressbar=None, num_workers=None, display=True, toolkit=None, get_intensity=False, **kwargs*)

Find peaks in a 2D signal.

Function to locate the positive peaks in an image using various, user specified, methods. Returns a structured array containing the peak positions.

Parameters**method**

[`str`] Select peak finding algorithm to implement. Available methods are:

- 'local_max' - simple local maximum search using the `skimage.feature.peak_local_max()` function
- 'max' - simple local maximum search using the `find_peaks_max()`.
- 'minmax' - finds peaks by comparing maximum filter results with minimum filter, calculates centers of mass. See the `find_peaks_minmax()` function.
- 'zaefferer' - based on gradient thresholding and refinement by local region of interest optimisation. See the `find_peaks_zaefferer()` function.

⁰ Schaffer, Bernhard, Werner Grogger, and Gerald Kothleitner. "Automated Spatial Drift Correction for EFTEM Image Series." *Ultramicroscopy* 102, no. 1 (December 2004): 27–36.

- ‘stat’ - based on statistical refinement and difference with respect to mean intensity. See the `find_peaks_stat()` function.
- ‘laplacian_of_gaussian’ - a blob finder using the laplacian of Gaussian matrices approach. See the `find_peaks_log()` function.
- ‘difference_of_gaussian’ - a blob finder using the difference of Gaussian matrices approach. See the `find_peaks_dog()` function.
- ‘template_matching’ - A cross correlation peakfinder. This method requires providing a template with the `template` parameter, which is used as reference pattern to perform the template matching to the signal. It uses the `skimage.feature.match_template()` function and the peaks position are obtained by using `minmax` method on the template matching result.

interactive

[bool] If True, the method parameter can be adjusted interactively. If False, the results will be returned.

current_index

[bool] If True, the computation will be performed for the current index.

get_intensity

[bool] If True, the intensity of the peak will be returned as an additional column, the last one.

show_progressbar

[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

num_workers

[None or int] Number of worker used by dask. If None, default to dask default value.

display

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

****kwargs**

[dict] Keywords parameters associated with above methods, see the documentation of each method for more details.

Returns**peaks**

[BaseSignal or numpy.ndarray] numpy.ndarray if `current_index=True`. Ragged signal with shape (npeaks, 2) that contains the *x*, *y* pixel coordinates of peaks found in each image sorted first along *y* and then along *x*.

Notes

As a convenience, the ‘local_max’ method accepts the ‘distance’ and ‘threshold’ argument, which will be map to the ‘min_distance’ and ‘threshold_abs’ of the `skimage.feature.peak_local_max()` function.

```
plot(navigator='auto', plot_markers=True, autoscale='v', norm='auto', vmin=None, vmax=None,
      gamma=1.0, linthresh=0.01, linscale=0.1, scalebar=True, scalebar_color='white', axes_ticks=None,
      axes_off=False, axes_manager=None, no_nans=False, colorbar=True, centre_colormap='auto',
      min_aspect=0.1, navigator_kwds={}, **kwargs)
```

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

navigator

[str, None, or *BaseSignal* (or subclass).]

Allowed string values are ‘‘auto’’, ‘‘slider’’, and ‘‘spectrum’’.

- If 'auto':
 - If navigation_dimension > 0, a navigator is provided to explore the data.
 - If navigation_dimension is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If navigation_dimension is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If navigation_dimension is > 1, the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if navigation_dimension > 2, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”, and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.
 - For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If 'slider':
 - If navigation dimension > 0 a window with one slider per axis is raised to navigate the data.
- If 'spectrum':
 - If navigation_dimension > 0 the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the 'auto' option will be used instead.
- If None, no navigator will be provided.

Alternatively a *BaseSignal* (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the navigation_shape + signal_shape must be equal to the navigator_shape of

the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

axes_manager

[None or *AxesManager*] If None, the signal's axes_manager attribute is used.

plot_markers

[bool, default True] Plot markers added using *s.add_marker(marker, permanent=True)*. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[dict] Only for image navigator, additional keyword arguments for *matplotlib.pyplot.imshow()*.

colorbar

[bool, optional] If true, a colorbar is plotted for non-RGB images.

autoscale

[str, optional] The string must contain any combination of the 'x', 'y' and 'v' characters. If 'x' or 'y' are in the string, the corresponding axis limits are set to cover the full range of the data at a given position. If 'v' (for values) is in the string, the contrast of the image will be set automatically according to *vmin* and *vmax* when the data or navigation indices change. Default is 'v'.

norm

[str {"auto" | "linear" | "power" | "log" | "symlog"} or *matplotlib.colors.Normalize*] Set the norm of the image to display. If "auto", a linear scale is used except if when *power_spectrum=True* in case of complex data type. "symlog" can be used to display negative value on a negative scale - read *matplotlib.colors.SymLogNorm* and the *linthresh* and *linscale* parameter for more details.

vmin, vmax

[{scalar, str}, optional] *vmin* and *vmax* are used to normalise the displayed data. It can be a float or a string. If string, it should be formatted as 'xth', where 'x' must be an float in the [0, 100] range. 'x' is used to compute the x-th percentile of the data. See *numpy.percentile()* for more information.

gamma

[float, optional] Parameter used in the power-law normalisation when the parameter *norm="power"*. Read *matplotlib.colors.PowerNorm* for more details. Default value is 1.0.

linthresh

[float, optional] When used with *norm="symlog"*, define the range within which the plot is linear (to avoid having the plot go to infinity around zero). Default value is 0.01.

linscale

[float, optional] This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of powers of base to use for each half of the linear range. See *matplotlib.colors.SymLogNorm* for more details. Default value is 0.1.

scalebar

[bool, optional] If True and the units and scale of the x and y axes are the same a scale bar is plotted.

scalebar_color

[str, optional] A valid MPL color string; will be used as the scalebar color.

axes_ticks

[{None, bool}, optional] If True, plot the axes ticks. If None axes_ticks are only plotted when the scale bar is not plotted. If False the axes ticks are never plotted.

axes_off

[bool, default False]

no_nans

[bool, optional] If True, set nans to zero for plotting.

centre_colormap

[bool or "auto"] If True the centre of the color scheme is set to zero. This is specially useful when using diverging color schemes. If "auto" (default), diverging color schemes are automatically centred.

min_aspect

[float, optional] Set the minimum aspect ratio of the image and the figure. To keep the image in the aspect limit the pixels are made rectangular.

****kwargs**

[dict] Only when plotting an image: additional (optional) keyword arguments for `matplotlib.pyplot.imshow()`.

23.9 Base Classes

API of classes, which are not part of the *hyperspy.api* namespace but are inherited in HyperSpy classes. The signal classes are not expected to be instantiated by users but their methods, which are used by other classes, are documented here.

23.9.1 Axes

<i>BaseDataAxis</i> ([index_in_array, name, units, ...])	Parent class defining common attributes for all DataAxis classes.
<i>DataAxis</i> ([index_in_array, name, units, ...])	DataAxis class for a non-uniform axis defined through an axis array.
<i>FunctionalDataAxis</i> (expression[, x, ...])	DataAxis class for a non-uniform axis defined through an expression.
<i>UniformDataAxis</i> ([index_in_array, name, ...])	DataAxis class for a uniform axis defined through a scale, an offset and a size.
<i>AxesManager</i> (axes_list)	Contains and manages the data axes.
<i>UnitConversion</i> ([units, scale, offset])	Parent class containing unit conversion functionalities of Uniform Axis.

class `hyperspy.axes.AxesManager`(*axes_list*)

Bases: `HasTraits`

Contains and manages the data axes.

It supports indexing, slicing, subscripting and iteration. As an iterator, iterate over the navigation coordinates returning the current indices. It can only be indexed and sliced to access the DataAxis objects that it contains. Standard indexing and slicing follows the "natural order" as in Signal, i.e. [nX, nY, ...,sX, sY,...] where *n* indicates a navigation axis and *s* a signal axis. In addition, AxesManager supports indexing using complex numbers *a + bj*, where *b* can be one of 0, 1, 2 and 3 and *a* a valid index. If *b* is 3, AxesManager is indexed using

the order of the axes in the array. If `b` is 1(2), indexes only the navigation(signal) axes in the natural order. In addition `AxesManager` supports subscription using axis name.

Examples

Create a spectrum with random data

```
>>> s = hs.signals.Signal1D(np.random.random((2,3,4,5)))
>>> s.axes_manager
<Axes manager, axes: (4, 3, 2|5)>
      Name | size | index | offset | scale | units
===== | ===== | ===== | ===== | ===== | =====
<undefined> | 4 | 0 | 0 | 1 | <undefined>
<undefined> | 3 | 0 | 0 | 1 | <undefined>
<undefined> | 2 | 0 | 0 | 1 | <undefined>
----- | ----- | ----- | ----- | ----- | -----
<undefined> | 5 | 0 | 0 | 1 | <undefined>
>>> s.axes_manager[0]
<Unnamed 0th axis, size: 4, index: 0>
>>> s.axes_manager[3j]
<Unnamed 2nd axis, size: 2, index: 0>
>>> s.axes_manager[1j]
<Unnamed 0th axis, size: 4, index: 0>
>>> s.axes_manager[2j]
<Unnamed 3rd axis, size: 5>
>>> s.axes_manager[1].name = "y"
>>> s.axes_manager["y"]
<y axis, size: 3, index: 0>
>>> for i in s.axes_manager:
...     print(i, s.axes_manager.indices)
(0, 0, 0) (0, 0, 0)
(1, 0, 0) (1, 0, 0)
(2, 0, 0) (2, 0, 0)
(3, 0, 0) (3, 0, 0)
(3, 1, 0) (3, 1, 0)
(2, 1, 0) (2, 1, 0)
(1, 1, 0) (1, 1, 0)
(0, 1, 0) (0, 1, 0)
(0, 2, 0) (0, 2, 0)
(1, 2, 0) (1, 2, 0)
(2, 2, 0) (2, 2, 0)
(3, 2, 0) (3, 2, 0)
(3, 2, 1) (3, 2, 1)
(2, 2, 1) (2, 2, 1)
(1, 2, 1) (1, 2, 1)
(0, 2, 1) (0, 2, 1)
(0, 1, 1) (0, 1, 1)
(1, 1, 1) (1, 1, 1)
(2, 1, 1) (2, 1, 1)
(3, 1, 1) (3, 1, 1)
(3, 0, 1) (3, 0, 1)
(2, 0, 1) (2, 0, 1)
(1, 0, 1) (1, 0, 1)
```

(continues on next page)

(continued from previous page)

`(0, 0, 1) (0, 0, 1)`**Attributes****signal_axes, navigation_axes**`[list]` Contain the corresponding `DataAxis` objects**coordinates, indices, iterpath****property axes_are_aligned_with_data**

Verify if the data axes are aligned with the signal axes.

When the data are aligned with the axes the axes order in `self._axes` is `[nav_n, nav_n-1, ..., nav_0, sig_m, sig_m-1, ..., sig_0]`.**Returns****aligned**`[bool]`**convert_units**(*axes=None, units=None, same_units=True, factor=0.25*)

Convert the scale and the units of the selected axes. If the unit of measure is not supported by the pint library, the scale and units are not changed.

Parameters**axes**`[int, str, iterable of DataAxis or None, default None]` Convert to a convenient scale and units on the specified axis. If `int`, the axis can be specified using the index of the axis in `axes_manager`. If string, argument can be "navigation" or "signal" to select the navigation or signal axes. The axis name can also be provided. If `None`, convert all axes.**units**`[list of str, str or None, default None]` If list, the selected axes will be converted to the provided units. If string, the navigation or signal axes will be converted to the provided units. If `None`, the scale and the units are converted to the appropriate scale and units to avoid displaying scalebar with >3 digits or too small number. This can be tweaked by the `factor` argument.**same_units**`[bool]` If `True`, force to keep the same units if the units of the axes differs. It only applies for the same kind of axis, "navigation" or "signal". By default the converted unit of the first axis is used for all axes. If `False`, convert all axes individually.**factor**`[float (default: 0.25)]` 'factor' is an adjustable value used to determine the prefix of the units. The product `factor * scale * size` is passed to the pint `to_compact` method to determine the prefix.

Notes

Requires a uniform axis.

property coordinates

Get and set the current coordinates, if the navigation dimension is not 0. If the navigation dimension is 0, it raises `AttributeError` when attempting to set its value.

create_axes(*axes_list*)

Given a list of either axes dictionaries, these are added to the `AxesManager`. In case dictionaries defining the axes properties are passed, the `DataAxis`, `UniformDataAxis`, `FunctionalDataAxis` instances are first created.

The index of the axis in the array and in the `_axes` lists can be defined by the `index_in_array` keyword if given for all axes. Otherwise, it is defined by their index in the list.

Parameters

axes_list

[list of dict] The list of axes to create.

gui(*display=True*, *toolkit=None*, ***kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

gui_navigation_sliders(*title=""*, *display=True*, *toolkit=None*)

Navigation sliders to control the index of the navigation axes.

Parameters

title: str

%s

%s

property indices

Get and set the current indices, if the navigation dimension is not 0. If the navigation dimension is 0, it raises `AttributeError` when attempting to set its value.

property iterpath

Sets the order of iterating through the indices in the navigation dimension. Can be either “flyback” or “serpentine”, or an iterable of navigation indices.

key_navigator(*event*)

Set hotkeys for controlling the indices of the navigator plot

property navigation_axes

The navigation axes as a tuple.

property navigation_dimension

The dimension of the navigation space.

property navigation_extent

The low and high values of the navigation axes.

property navigation_shape

The shape of the navigation space.

property navigation_size

The size of the navigation space.

remove(*axes*)

Remove one or more axes

set_axis(*axis, index_in_axes_manager*)

Replace an axis of current signal with one given in argument.

Parameters**axis**

[*BaseDataAxis*] The axis to replace the current axis with.

index_in_axes_manager

[*int*] The index of the axis in current signal to replace with the axis passed in argument.

property signal_axes

The signal axes as a tuple.

property signal_dimension

The dimension of the signal space.

property signal_extent

The low and high values of the signal axes.

property signal_shape

The shape of the signal space.

property signal_size

The size of the signal space.

switch_iterpath(*iterpath=None*)

Context manager to change iterpath. The original iterpath is restored when exiting the context.

Parameters**iterpath**

[*str*, optional] The iterpath to use. The default is None.

Yields

None.

Examples

```
>>> s = hs.signals.Signal1D(np.arange(2*3*4).reshape([3, 2, 4]))
>>> with s.axes_manager.switch_iterpath('serpentine'):
...     for indices in s.axes_manager:
...         print(indices)
(0, 0)
(1, 0)
(1, 1)
(0, 1)
(0, 2)
(1, 2)
```

update_axes_attributes_from(*axes*, *attributes=None*)

Update the axes attributes to match those given.

The axes are matched by their index in the array. The purpose of this method is to update multiple axes triggering *any_axis_changed* only once.

Parameters

axes: iterable of :class:`~hyperspy.axes.DataAxis`.

The axes to copy the attributes from.

attributes: iterable of strings.

The attributes to copy.

class hyperspy.axes.**BaseDataAxis**(*index_in_array=None*, *name=None*, *units=None*, *navigate=False*, *is_binned=False*, ***kwargs*)

Bases: `HasTraits`

Parent class defining common attributes for all DataAxis classes.

Parameters

name

[`str`, optional] Name string by which the axis can be accessed. *<undefined>* if not set.

units

[`str`, optional] String for the units of the axis vector. *<undefined>* if not set.

navigate

[`bool`, optional] True for a navigation axis. Default False (signal axis).

is_binned

[`bool`, optional] True if data along the axis is binned. Default False.

convert_to_uniform_axis()

Convert to an uniform axis.

gui(*display=True*, *toolkit=None*, ***kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[`bool`] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[`str`, iterable of `str` or `None`] If `None` (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

update_from(*axis*, *attributes*)

Copy values of specified axes fields from the passed AxesManager.

Parameters**axis**

[`BaseDataAxis`] The instance to use as a source for values.

attributes

[iterable of `str`] The name of the attribute to update. If the attribute does not exist in either of the AxesManagers, an `AttributeError` will be raised.

Returns**bool**

True if any changes were made, otherwise False.

value2index(*value*, *rounding*=<built-in function round>)

Return the closest index/indices to the given value(s) if between the axis limits.

Parameters**value**

[`float` or `numpy.ndarray`]

rounding

[`callable()`] Handling of values between two axis points:

- If `rounding=round`, use round-half-away-from-zero strategy to find closest value.
- If `rounding=math.floor`, round to the next lower value.
- If `rounding=math.ceil`, round to the next higher value.

Returns

`int` or `numpy array`

Raises**ValueError**

If `value` is out of bounds or contains out of bounds values (array). If `value` is `NaN` or contains `NaN` values (array).

value_range_to_indices(*v1*, *v2*)

Convert the given range to index range.

When a value is out of the axis limits, the endpoint is used instead. *v1* must be preceding *v2* in the axis values

- if the axis scale is positive, it means $v1 < v2$
- if the axis scale is negative, it means $v1 > v2$

Parameters**v1, v2**

[`float`] The end points of the interval in the axis units.

Returns

i2, i2

[float] The indices corresponding to the interval [v1, v2]

class hyperspy.axes.**DataAxis**(*index_in_array=None, name=None, units=None, navigate=False, is_binned=False, axis=[1], **kwargs*)

Bases: [BaseDataAxis](#)

DataAxis class for a non-uniform axis defined through an axis array.

The most flexible type of axis, where the axis points are directly given by an array named `axis`. As this can be any array, the property `is_uniform` is automatically set to `False`.

Parameters

axis

[numpy array or list] The array defining the axis points.

Examples

Sample dictionary for a *DataAxis*:

```
>>> dict0 = {'axis': np.arange(11)**2}
>>> s = hs.signals.Signal1D(np.ones(12), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'DataAxis',
 'name': None,
 'units': None,
 'navigate': False,
 'is_binned': False,
 'axis': array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81, 100])}
```

crop(*start=None, end=None*)

Crop the axis in place.

Parameters

start

[int, float, or None] The beginning of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

end

[int, float, or None] The end of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

update_axis()

Set the value of an axis. The axis values need to be ordered.

Raises

ValueError

If the axis values are not ordered.

update_from(*axis, attributes=None*)

Copy values of specified axes fields from the passed AxesManager.

Parameters

axis

[*DataAxis*] The instance to use as a source for values.

attributes

[iterable of *str*] The name of the attribute to update. If the attribute does not exist in either of the AxesManagers, an *AttributeError* will be raised. If *None*, units will be updated.

Returns

bool

True if any changes were made, otherwise False.

```
class hyperspy.axes.FunctionalDataAxis(expression, x=None, index_in_array=None, name=None,
                                       units=None, navigate=False, size=1, is_binned=False,
                                       **parameters)
```

Bases: *BaseDataAxis*

DataAxis class for a non-uniform axis defined through an expression.

A *FunctionalDataAxis* is defined based on an expression that is evaluated to yield the axis points. The *expression* is a function defined as a string using the *SymPy* text expression format. An example would be `expression = a / x + b`. Any variables in the expression, in this case *a* and *b* must be defined as additional attributes of the axis. The property `is_uniform` is automatically set to `False`.

x itself is an instance of *BaseDataAxis*. By default, it will be a *UniformDataAxis* with `offset = 0` and `scale = 1` of the given size. However, it can also be initialized with custom `offset` and `scale` values. Alternatively, it can be a non-uniform *DataAxis*.

Parameters

expression: str

SymPy mathematical expression defining the axis.

x

[*BaseDataAxis*] Defines *x*-values at which *expression* is evaluated.

Examples

Sample dictionary for a *FunctionalDataAxis*:

```
>>> dict0 = {'expression': 'a / (x + 1) + b', 'a': 100, 'b': 10, 'size': 500}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'FunctionalDataAxis',
 'name': None,
 'units': None,
 'navigate': False,
 'is_binned': False,
 'expression': 'a / (x + 1) + b',
 'size': 500,
 'x': {'_type': 'UniformDataAxis',
       'name': None,
       'units': None,
       'navigate': False,
       'is_binned': False,
```

(continues on next page)

(continued from previous page)

```
'size': 500,
'scale': 1.0,
'offset': 0.0},
'a': 100,
'b': 10}
```

convert_to_non_uniform_axis()

Convert to a non-uniform axis.

crop(*start=None, end=None*)

Crop the axis in place.

Parameters**start**

[[int](#), [float](#), or [None](#)] The beginning of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

end

[[int](#), [float](#), or [None](#)] The end of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start/end* is `None` the method crops from/to the low/high end of the axis.

update_from(*axis, attributes=None*)

Copy values of specified axes fields from the passed AxesManager.

Parameters**axis**

[[FunctionalDataAxis](#)] The instance to use as a source for values.

attributes

[[iterable](#) of [str](#) or [None](#)] A list of the name of the attribute to update. If an attribute does not exist in either of the AxesManagers, an `AttributeError` will be raised. If `None`, the parameters of *expression* are updated.

Returns

A boolean indicating whether any changes were made.

```
class hyperspy.axes.UniformDataAxis(index_in_array=None, name=None, units=None, navigate=False,
                                     size=1, scale=1.0, offset=0.0, is_binned=False, **kwargs)
```

Bases: [BaseDataAxis](#), [UnitConversion](#)

DataAxis class for a uniform axis defined through a scale, an offset and a size.

The most common type of axis. It is defined by the `offset`, `scale` and `size` parameters, which determine the *initial value*, *spacing* and *length* of the axis, respectively. The actual axis array is automatically calculated from these three values. The `UniformDataAxis` is a special case of the `FunctionalDataAxis` defined by the function `scale * x + offset`.

Parameters**offset**

[[float](#)] The first value of the axis vector.

scale
[float] The spacing between axis points.

size
[int] The number of points in the axis.

Examples

Sample dictionary for a *UniformDataAxis*:

```
>>> dict0 = {'offset': 300, 'scale': 1, 'size': 500}
>>> s = hs.signals.Signal1D(np.ones(500), axes=[dict0])
>>> s.axes_manager[0].get_axis_dictionary()
{'_type': 'UniformDataAxis',
 'name': <undefined>,
 'units': <undefined>,
 'navigate': False,
 'size': 500,
 'scale': 1.0,
 'offset': 300.0}
```

crop(*start=None, end=None*)

Crop the axis in place.

Parameters

start

[int, float, or None] The beginning of the cropping interval. If type is int, the value is taken as the axis index. If type is float the index is calculated using the axis calibration. If *start/end* is None the method crops from/to the low/high end of the axis.

end

[int, float, or None] The end of the cropping interval. If type is int, the value is taken as the axis index. If type is float the index is calculated using the axis calibration. If *start/end* is None the method crops from/to the low/high end of the axis.

update_from(*axis, attributes=None*)

Copy values of specified axes fields from the passed AxesManager.

Parameters

axis

[*UniformDataAxis*] The *UniformDataAxis* instance to use as a source for values.

attributes

[iterable of str or None] The name of the attribute to update. If the attribute does not exist in either of the AxesManagers, an *AttributeError* will be raised. If *None*, *scale*, *offset* and *units* are updated.

Returns

A boolean indicating whether any changes were made.

value2index(*value, rounding=<built-in function round>*)

Return the closest index/indices to the given value(s) if between the axis limits.

Parameters

value

[`float`, `str`, `numpy.ndarray`] If string, should either be a calibrated unit like “20nm” or a relative slicing like “rel0.2”.

rounding

[`callable()`] Handling of values intermediate between two axis points: If `rounding=round`, use python’s standard round-half-to-even strategy to find closest value. If `rounding=math.floor`, round to the next lower value. If `rounding=math.ceil`, round to the next higher value.

Returns

`int` or `numpy.ndarray`

Raises**ValueError**

If value is out of bounds or contains out of bounds values (array). If value is NaN or contains NaN values (array). If value is incorrectly formatted str or contains incorrectly formatted str (array).

class `hyperspy.axes.UnitConversion`(*units=None, scale=1.0, offset=0.0*)

Bases: `object`

Parent class containing unit conversion functionalities of Uniform Axis.

Parameters**offset**

[`float`] The first value of the axis vector.

scale

[`float`] The spacing between axis points.

size

[`int`] The number of points in the axis.

convert_to_units(*units=None, inplace=True, factor=0.25*)

Convert the scale and the units of the current axis. If the unit of measure is not supported by the pint library, the scale and units are not modified.

Parameters**units**

[`{str | None}`] Default = None If str, the axis will be converted to the provided units. If “*auto*”, automatically determine the optimal units to avoid using too large or too small numbers. This can be tweaked by the *factor* argument.

inplace

[`bool`] If *True*, convert the axis in place. if *False* return the *scale*, *offset* and *units*.

factor

[`float` (default: 0.25)] ‘factor’ is an adjustable value used to determine the prefix of the units. The product *factor* * *scale* * *size* is passed to the `pint.to_compact` method to determine the prefix.

23.9.2 Events

class `hyperspy.events.Event` (*doc=""*, *arguments=None*)

Bases: `object`

Events class

Parameters

doc

[`str`] Optional docstring for the new Event.

arguments

[`iterable`] Pass to define the arguments of the `trigger()` function. Each element must either be an argument name, or a tuple containing the argument name and the argument's default value.

Examples

```
>>> from hyperspy.events import Event
>>> Event()
<hyperspy.events.Event: set()>
>>> Event(doc="This event has a docstring!").__doc__
'This event has a docstring!'
>>> e1 = Event()
>>> e2 = Event(arguments=('arg1', ('arg2', None)))
>>> e1.trigger(arg1=12, arg2=43, arg3='str', arg4=4.3) # Can trigger with whatever
>>> e2.trigger(arg1=11, arg2=22, arg3=3.4)
Traceback (most recent call last):
...
TypeError: trigger() got an unexpected keyword argument 'arg3'
```

connect (*function*, *kwargs='all'*)

Connects a function to the event.

Parameters

function

[`callable()`] The function to call when the event triggers.

kwargs

[`tuple` or `list`, `dict`, `str` {'all' | ``'auto'``, default "all"] If "all", all the trigger keyword arguments are passed to the function. If a list or tuple of strings, only those keyword arguments that are in the tuple or list are passed. If empty, no keyword argument is passed. If dictionary, the keyword arguments of trigger are mapped as indicated in the dictionary. For example, {"a": "b"} maps the trigger argument "a" to the function argument "b".

See also:

[`disconnect`](#)

property `connected`

Connected functions.

disconnect(*function*)

Disconnects a function from the event. The passed function will be disconnected irregardless of which ‘nargs’ argument was passed to connect().

If you only need to temporarily prevent a function from being called, single callback suppression is supported by the *suppress_callback* context manager.

Parameters

function: **function**

return_connection_kwargs: **bool, default False**

If True, returns the kwargs that would reconnect the function as it was.

See also:

[*connect*](#)

[*suppress_callback*](#)

suppress()

Use this function with a ‘with’ statement to temporarily suppress all events in the container. When the ‘with’ lock completes, the old suppression values will be restored.

See also:

[*suppress_callback*](#)

[*Events.suppress*](#)

Examples

```
>>> with obj.events.myevent.suppress():
...     # These would normally both trigger myevent:
...     obj.val_a = a
...     obj.val_b = b
```

Trigger manually once: >>> obj.events.myevent.trigger() # doctest: +SKIP

suppress_callback(*function*)

Use this function with a ‘with’ statement to temporarily suppress a single callback from being called. All other connected callbacks will trigger. When the ‘with’ lock completes, the old suppression value will be restored.

See also:

[*suppress*](#)

[*Events.suppress*](#)

Examples

```
>>> with obj.events.myevent.suppress_callback(f):
...     # Events will trigger as normal, but `f` will not be called
...     obj.val_a = a
...     obj.val_b = b
>>> # Here, `f` will be called as before:
>>> obj.events.myevent.trigger()
```

trigger(**kwargs)

Triggers the event. If the event is suppressed, this does nothing. Otherwise it calls all the connected functions with the arguments as specified when connected.

See also:

[suppress](#)
[suppress_callback](#)
[Events.suppress](#)

class hyperspy.events.EventSuppressor(*to_suppress)

Bases: [object](#)

Object to enforce a variety of suppression types simultaneously

Targets to be suppressed can be added by the function *add()*, or given in the constructor. Valid targets are:

- *Event*: The entire Event will be suppressed
- *Events*: All events in th container will be suppressed
- (Event, callback): The callback will be suppressed in Event
- (Events, callback): The callback will be suppressed in each event in Events where it is connected.
- Any iterable collection of the above target types

Examples

```
>>> es = EventSuppressor((event1, callback1), (event1, callback2))
>>> es.add(event2, callback2)
>>> es.add(event3)
>>> es.add(events_container1)
>>> es.add(events_container2, callback1)
>>> es.add(event4, (events_container3, callback2))
```

```
>>> with es.suppress():
...     do_something()
```

add(*to_suppress)

Add one or more targets to be suppressed

Valid targets are:

- *Event*: The entire Event will be suppressed
- *Events*: All events in the container will be suppressed
- (Event, callback): The callback will be suppressed in Event

- (Events, callback): The callback will be suppressed in each event in Events where it is connected.
- Any iterable collection of the above target types

suppress()

Use this function with a 'with' statement to temporarily suppress all events added. When the 'with' lock completes, the old suppression values will be restored.

See also:

Events.suppress

Event.suppress

Event.suppress_callback

class hyperspy.events.Events

Bases: `object`

Events container.

All available events are attributes of this class.

suppress()

Use this function with a 'with' statement to temporarily suppress all callbacks of all events in the container. When the 'with' lock completes, the old suppression values will be restored.

See also:

Event.suppress

Event.suppress_callback

Examples

```
>>> with obj.events.suppress():
...     # Any events triggered by assignments are prevented:
...     obj.val_a = a
...     obj.val_b = b
>>> # Trigger one event instead:
>>> obj.events.values_changed.trigger()
```

23.9.3 Machine Learning

class hyperspy.learn.mva.LearningResults

Bases: `object`

Stores the parameters and results from a decomposition.

crop_decomposition_dimension(*n*, *compute=False*)

Crop the score matrix up to the given number.

It is mainly useful to save memory and reduce the storage size

Parameters

n

[int] Number of components to keep.

compute

[`bool`, default `False`] If True and the decomposition results are lazy, also compute the results.

load(*filename*)

Load the results of a previous decomposition and demixing analysis.

Parameters**filename**

[`str`] Path to load the results from.

save(*filename*, *overwrite=None*)

Save the result of the decomposition and demixing analysis.

Parameters**filename**

[`str`] Path to save the results to.

overwrite

[`{True, False, None}`, default `None`] If True, overwrite the file if it exists. If None (default), prompt user if file exists.

summary()

Summarize the decomposition and demixing parameters.

Returns**str**

String summarizing the learning parameters.

`hyperspy.learn.ml pca.ml pca(X, varX, output_dimension, svd_solver='auto', tol=1e-10, max_iter=50000, **kwargs)`

Performs maximum likelihood PCA with missing data and/or heteroskedastic noise.

Standard PCA based on a singular value decomposition (SVD) approach assumes that the data is corrupted with Gaussian, or homoskedastic noise. For many applications, this assumption does not hold. For example, count data from EDS-TEM experiments is corrupted by Poisson noise, where the noise variance depends on the underlying pixel value. Rather than scaling or transforming the data to approximately “normalize” the noise, MLPCA instead uses estimates of the data variance to perform the decomposition.

This function is a transcription of a MATLAB code obtained from [Andrews1997].

Read more in the *User Guide*.

Parameters**X**

[`numpy.ndarray`] Matrix of observations with shape (m, n).

varX

[`numpy.ndarray`] Matrix of variances associated with X (zeros for missing measurements).

output_dimension

[`int`] The model dimensionality.

svd_solver

[`{"auto", "full", "arpack", "randomized"}`, default `"auto"`]

If auto:

The solver is selected by a default policy based on `data.shape` and `output_dimension`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

If full:

run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing

If arpack:

use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly $0 < output_dimension < min(data.shape)$

If randomized:

use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

tol

[float] Tolerance of the stopping condition.

max_iter

[int] Maximum number of iterations before exiting without convergence.

Returns**numpy.ndarray**

The pseudo-SVD parameters.

float

Value of the objective function.

References

[Andrews1997]

```
class hyperspy.learn.ornmf.ORNMF(rank, store_error=False, lambda1=1.0, kappa=1.0, method='PGD',
                                subspace_learning_rate=1.0, subspace_momentum=0.5,
                                random_state=None)
```

Bases: `object`

Performs Online Robust NMF with missing or corrupted data.

The ORNMF code is based on a transcription of the online proximal gradient descent (PGD) algorithm MATLAB code obtained from the authors of [Zhao2016]. It has been updated to also include L2-normalization cost function that is able to deal with sparse corruptions and/or outliers slightly faster (please see ORPCA implementation for details). A further modification has been made to allow for a changing subspace W , where $X \approx WH^T + E$ in the ORNMF framework.

Read more in the *User Guide*.

References

[Zhao2016]

Creates Online Robust NMF instance that can learn a representation.

Parameters

rank

[[int](#)] The rank of the representation (number of components/factors)

store_error

[[bool](#), default `False`] If True, stores the sparse error matrix.

lambda1

[[float](#)] Nuclear norm regularization parameter.

kappa

[[float](#)] Step-size for projection solver.

method

[{ 'PGD', 'RobustPGD', 'MomentumSGD' }, default 'PGD']

- 'PGD' - Proximal gradient descent
- 'RobustPGD' - Robust proximal gradient descent
- 'MomentumSGD' - Stochastic gradient descent with momentum

subspace_learning_rate

[[float](#)] Learning rate for the 'MomentumSGD' method. Should be a float > 0.0

subspace_momentum

[[float](#)] Momentum parameter for 'MomentumSGD' method, should be a float between 0 and 1.

random_state

[[None](#) or [int](#) or [RandomState](#), default `None`] Used to initialize the subspace on the first iteration. See [numpy.random.default_rng\(\)](#) for more information.

finish()

Return the learnt factors and loadings.

fit(*X*, *batch_size=None*)

Learn NMF components from the data.

Parameters

X

[[array_like](#)] [*n_samples* x *n_features*] matrix of observations or an iterator that yields samples, each with *n_features* elements.

batch_size

[{[None](#), [int](#)}] If not None, learn the data in batches, each of *batch_size* samples or less.

project(*X*, *return_error=False*)

Project the learnt components on the data.

Parameters

X

[[array_like](#)] The matrix of observations with shape (*n_samples*, *n_features*) or an iterator that yields *n_samples*, each with *n_features* elements.

return_error

[bool, default False] If True, returns the sparse error matrix as well. Otherwise only the weights (loadings)

```
hyperspy.learn.ornmf.ornmf(X, rank, store_error=False, project=False, batch_size=None, lambda1=1.0, kappa=1.0, method='PGD', subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None)
```

Perform online, robust NMF on the data X.

This is a wrapper function for the ORNMF class.

Parameters**X**

[numpy.ndarray] The [n_samples, n_features] input data.

rank

[int] The rank of the representation (number of components/factors)

store_error

[bool, default False] If True, stores the sparse error matrix.

project

[bool, default False] If True, project the data X onto the learnt model.

batch_size

[None or int, default None] If not None, learn the data in batches, each of batch_size samples or less.

lambda1

[float, default 1.0] Nuclear norm regularization parameter.

kappa

[float, default 1.0] Step-size for projection solver.

method

[{'PGD', 'RobustPGD', 'MomentumSGD'}, default 'PGD']

- 'PGD' - Proximal gradient descent
- 'RobustPGD' - Robust proximal gradient descent
- 'MomentumSGD' - Stochastic gradient descent with momentum

subspace_learning_rate

[float, default 1.0] Learning rate for the 'MomentumSGD' method. Should be a float > 0.0

subspace_momentum

[float, default 0.5] Momentum parameter for 'MomentumSGD' method, should be a float between 0 and 1.

random_state

[None or int or RandomState, default None] Used to initialize the subspace on the first iteration.

Returns**Xhat**

[numpy.ndarray] The non-negative matrix with shape (n_features x n_samples). Only returned if store_error is True.

Ehat

[`numpy.ndarray`] The sparse error matrix with shape (n_features x n_samples). Only returned if store_error is True.

W

[`numpy.ndarray`] The non-negative factors matrix with shape (n_features, rank).

H

[`numpy.ndarray`] The non-negative loadings matrix with shape (rank, n_samples).

`hyperspy.learn.orthomax.orthomax(A, gamma=1.0, tol=1.4901e-07, max_iter=256)`

Calculate orthogonal rotations for a matrix of factors or loadings from PCA.

When gamma=1.0, this is known as varimax rotation, which finds a rotation matrix W that maximizes the variance of the squared components of $A @ W$. The rotation matrix preserves orthogonality of the components.

Taken from metpy.

Parameters**A**

[`numpy array`] Input data to unmix

gamma

[`float`] If gamma in range [0, 1], use SVD approach, otherwise solve with a sequence of bivariate rotations.

tol

[`float`] Tolerance of the stopping condition.

max_iter

[`int`] Maximum number of iterations before exiting without convergence.

Returns**B**

[`numpy array`] Rotated data matrix

W

[`numpy array`] The unmixing matrix

`class hyperspy.learn.rpca.ORPCA(rank, store_error=False, lambda1=0.1, lambda2=1.0, method='BCD', init='qr', training_samples=10, subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None)`

Bases: `object`

Performs Online Robust PCA with missing or corrupted data.

The ORPCA code is based on a transcription of MATLAB code from [Feng2013]. It has been updated to include a new initialization method based on a QR decomposition of the first n “training” samples of the data. A stochastic gradient descent (SGD) solver is also implemented, along with a MomentumSGD solver for improved convergence and robustness with respect to local minima. More information about the gradient descent methods and choosing appropriate parameters can be found in [Ruder2016].

Read more in the *User Guide*.

References

[Feng2013], [Ruder2016]

Creates Online Robust PCA instance that can learn a representation.

Parameters

rank

[`int`] The rank of the representation (number of components/factors)

store_error

[`bool`, default `False`] If True, stores the sparse error matrix.

lambda1

[`float`, default 0.1] Nuclear norm regularization parameter.

lambda2

[`float`, default 1.0] Sparse error regularization parameter.

method

[{'CF', 'BCD', 'SGD', 'MomentumSGD'}, default 'BCD']

- 'CF' - Closed-form solver
- 'BCD' - Block-coordinate descent
- 'SGD' - Stochastic gradient descent
- 'MomentumSGD' - Stochastic gradient descent with momentum

init

[`numpy.ndarray`, {'qr', 'rand'}, default 'qr']

- 'qr' - QR-based initialization
- 'rand' - Random initialization
- `numpy.ndarray` if the shape (n_features x rank)

training_samples

[`int`, default 10] Specifies the number of training samples to use in the 'qr' initialization.

subspace_learning_rate

[`float`, default 1.0] Learning rate for the 'SGD' and 'MomentumSGD' methods. Should be a float > 0.0

subspace_momentum

[`float`, default 0.5] Momentum parameter for 'MomentumSGD' method, should be a float between 0 and 1.

random_state

[`None`, `int` or `RandomState`, default `None`] Used to initialize the subspace on the first iteration.

finish(**kwargs)

Return the learnt factors and loadings.

fit(X, batch_size=None)

Learn RPCA components from the data.

Parameters

X

[array_like] The matrix of observations with shape (n_samples, n_features) or an iterator that yields samples, each with n_features elements.

batch_size

[None or int] If not None, learn the data in batches, each of batch_size samples or less.

project(X, return_error=False)

Project the learnt components on the data.

Parameters**X**

[array_like] The matrix of observations with shape (n_samples, n_features) or an iterator that yields n_samples, each with n_features elements.

return_error

[bool, default False] If True, returns the sparse error matrix as well. Otherwise only the weights (loadings)

`hyperspy.learn.rpca.orpca`(X, rank, store_error=False, project=False, batch_size=None, lambda1=0.1, lambda2=1.0, method='BCD', init='qr', training_samples=10, subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None, **kwargs)

Perform online, robust PCA on the data X.

This is a wrapper function for the ORPCA class.

Parameters**X**

[array_like] The matrix of observations with shape (n_features x n_samples) or an iterator that yields samples, each with n_features elements.

rank

[int] The rank of the representation (number of components/factors)

store_error

[bool, default False] If True, stores the sparse error matrix.

project

[bool, default False] If True, project the data X onto the learnt model.

batch_size

[None, int, default None] If not None, learn the data in batches, each of batch_size samples or less.

lambda1

[float, default 0.1] Nuclear norm regularization parameter.

lambda2

[float, default 1.0] Sparse error regularization parameter.

method

[{'CF', 'BCD', 'SGD', 'MomentumSGD'}, default 'BCD']

- 'CF' - Closed-form solver
- 'BCD' - Block-coordinate descent
- 'SGD' - Stochastic gradient descent

- 'MomentumSGD' - Stochastic gradient descent with momentum

init

[`numpy.ndarray`, {'qr', 'rand'}, default 'qr']

- 'qr' - QR-based initialization
- 'rand' - Random initialization
- `numpy.ndarray` if the shape [n_features x rank]

training_samples

[`int`, default 10] Specifies the number of training samples to use in the 'qr' initialization.

subspace_learning_rate

[`float`, default 1.0] Learning rate for the 'SGD' and 'MomentumSGD' methods. Should be a float > 0.0

subspace_momentum

[`float`, default 0.5] Momentum parameter for 'MomentumSGD' method, should be a float between 0 and 1.

random_state

[`None` or `int` or `RandomState`, default `None`] Used to initialize the subspace on the first iteration.

Returns**`numpy.ndarray`**

- If `project` is `True`, returns the low-rank factors and loadings only
- Otherwise, returns the low-rank and sparse error matrices, as well as the results of a singular value decomposition (SVD) applied to the low-rank matrix.

`hyperspy.learn.rpca.rpca_godec(X, rank, lambda1=None, power=0, tol=0.001, maxiter=1000, random_state=None, **kwargs)`

Perform Robust PCA with missing or corrupted data, using the GoDec algorithm.

Decomposes a matrix $Y = X + E$, where X is low-rank and E is a sparse error matrix. This algorithm is based on the Matlab code from [Zhou2011]. See code here: <https://sites.google.com/site/godecomposition/matrix/artifact-1>

Read more in the *User Guide*.

Parameters**X**

[`numpy.ndarray`] The matrix of observations with shape (n_features, n_samples)

rank

[`int`] The model dimensionality.

lambda1

[`None` or `float`] Regularization parameter. If `None`, set to $1 / \sqrt{n_features}$

power

[`int`, default 0] The number of power iterations used in the initialization

tol

[`float`, default $1e-3$] Convergence tolerance

maxiter

[`int`, default 1000] Maximum number of iterations

random_state

[None, int or RandomState, default None] Used to initialize the subspace on the first iteration.

Returns**Xhat**

[numpy.ndarray] The low-rank matrix with shape (n_features, n_samples)

Ehat

[numpy.ndarray] The sparse error matrix with shape (n_features, n_samples)

U, S, V

[numpy.ndarray] The results of an SVD on Xhat

References

[Zhou2011]

hyperspy.learn.svd_pca.**svd_flip_signs**(u, v, u_based_decision=True)

Sign correction to ensure deterministic output from SVD.

Adjusts the columns of u and the rows of v such that the loadings in the columns in u that are largest in absolute value are always positive.

Parameters**u, v**

[numpy.ndarray] u and v are the outputs of a singular value decomposition.

u_based_decision

[bool, default True] If True, use the columns of u as the basis for sign flipping. Otherwise, use the rows of v. The choice of which variable to base the decision on is generally algorithm dependent.

Returns**u, v**

[numpy.ndarray] Adjusted outputs with same dimensions as inputs.

hyperspy.learn.svd_pca.**svd_pca**(data, output_dimension=None, svd_solver='auto', centre=None, auto_transpose=True, svd_flip=True, **kwargs)

Perform PCA using singular value decomposition (SVD).

Read more in the *User Guide*.

Parameters**data**

[numpy array] MxN array of input data (M features, N samples)

output_dimension

[None or int] Number of components to keep/calculate

svd_solver

[{"auto", "full", "arnpack", "randomized"}, default "auto"]

If auto:

The solver is selected by a default policy based on *data.shape* and *output_dimension*: if the input data is larger than 500x500 and the number of components to extract is

lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

If full:

run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing

If arpack:

use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly $0 < output_dimension < min(data.shape)$

If randomized:

use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

centre

[{None, “navigation”, “signal”}, default None]

- If None, the data is not centered prior to decomposition.
- If “navigation”, the data is centered along the navigation axis.
- If “signal”, the data is centered along the signal axis.

auto_transpose

[bool, default True] If True, automatically transposes the data to boost performance.

svd_flip

[bool, default True] If True, adjusts the signs of the loadings and factors such that the loadings that are largest in absolute value are always positive. See `svd_flip_signs()` for more details.

Returns**factors**

[numpy.ndarray]

loadings

[numpy.ndarray]

explained_variance

[numpy.ndarray]

mean

[numpy.ndarray or None] None if centre is None

`hyperspy.learn.svd_pca.svd_solve(data, output_dimension=None, svd_solver='auto', svd_flip=True, u_based_decision=True, **kwargs)`

Apply singular value decomposition to input data.

Parameters**data**

[numpy.ndarray] Input data array with shape (m, n)

output_dimension

[None or int] Number of components to keep/calculate

svd_solver

[{“auto”, “full”, “arpack”, “randomized”}, default “auto”]

- If "auto": The solver is selected by a default policy based on *data.shape* and *output_dimension*: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient "randomized" method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.
- If "full": Run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing
- If "arpack": Use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly $0 < output_dimension < min(data.shape)$
- If "randomized": Use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

svd_flip

[bool, default True] If True, adjusts the signs of the loadings and factors such that the loadings that are largest in absolute value are always positive. See `svd_flip_signs()` for more details.

u_based_decision

[bool, default True] If True, and `svd_flip` is True, use the columns of *u* as the basis for sign-flipping. Otherwise, use the rows of *v*. The choice of which variable to base the decision on is generally algorithm dependent.

Returns

U, S, V

[numpy.ndarray] Output of SVD such that $X = U*S*V.T$

`hyperspy.learn.whitening.whiten_data(X, centre=True, method='PCA', epsilon=1e-10)`

Centre and whiten the data *X*.

A whitening transformation is used to decorrelate the variables, such that the new covariance matrix of the whitened data is the identity matrix.

If *X* is a random vector with non-singular covariance matrix *C*, and *W* is a whitening matrix satisfying $W^T W = C^{-1}$, then the transformation $Y = W X$ will yield a whitened random vector *Y* with unit diagonal covariance. In ZCA whitening, the matrix $W = C^{-1/2}$, while in PCA whitening, the matrix *W* is the eigensystem of *C*. More details can be found in [Kessy2015].

Parameters

X

[numpy.ndarray] The input data with shape (m, n).

centre

[bool, default True] If True, centre the data along the features axis. If False, do not centre the data.

method

[{"PCA", "ZCA"}] How to whiten the data. The default is PCA whitening.

epsilon

[float, default 1e-10] Small floating-point value to avoid divide-by-zero errors.

Returns

Y

[numpy.ndarray] The centred and whitened data with shape (m, n).

W

[numpy.ndarray] The whitening matrix with shape (n, n).

References

[Kessy2015]

23.9.4 Model

BaseModel

class `hyperspy.model.BaseModel`

Bases: `list`

Model and data fitting tools applicable to signals of both one and two dimensions.

Models of one-dimensional signals should use the *Model1D* and models of two-dimensional signals should use the *Model2D*.

A model is constructed as a linear combination of *components1D* or *components2D* that are added to the model using the *append()* or *extend()*. If needed, new components can be easily created using using *Expression* code of existing components as a template.

Once defined, the model can be fitted to the data using *fit()* or *multifit()*. Once the optimizer reaches the convergence criteria or the maximum number of iterations the new value of the component parameters are stored in the components.

It is possible to access the components in the model by their name or by the index in the model. An example is given at the end of this docstring.

See also:

hyperspy.models.model1d.Model1D, *hyperspy.models.model2d.Model2D*

Attributes

signal

[*BaseSignal*] The signal data to fit.

chisq

[*BaseSignal*] Chi-squared of the signal (or np.nan if not yet fit).

red_chisq

[*BaseSignal*] The Reduced chi-squared.

dof

[*BaseSignal*] Degrees of freedom of the signal (0 if not yet fit)

components

[*ModelComponents*] The components of the model are attributes of this class.

Methods

<code>append(thing)</code>	Add component to Model.
<code>extend(iterable)</code>	Append multiple components to the model.
<code>remove(thing)</code>	Remove component from model.
<code>set_signal_range_from_mask(mask)</code>	Use the signal ranges as defined by the mask
<code>fit([optimizer, loss_function, grad, ...])</code>	Fits the model to the experimental data.
<code>multifit([mask, fetch_only_fixed, autosave, ...])</code>	Fit the data to the model at all positions of the navigation dimensions.
<code>store_current_values()</code>	Store the parameters of the current coordinates into the <code>parameter.map</code> array and sets the <code>is_set</code> array attribute to True.
<code>fetch_stored_values([only_fixed, ...])</code>	Fetch the value of the parameters that have been previously stored in <code>parameter.map['values']</code> if <code>parameter.map['is_set']</code> is <code>True</code> for those indices.
<code>save_parameters2file(filename)</code>	Save the parameters array in binary format.
<code>load_parameters_from_file(filename)</code>	Loads the parameters array from a binary file written with the <code>save_parameters2file()</code> function.
<code>enable_plot_components()</code>	Enable interactive adjustment of the position of the components that have a well defined position.
<code>disable_plot_components()</code>	Disable interactive adjustment of the position of the components that have a well defined position.
<code>set_parameters_not_free([component_list, ...])</code>	Sets the parameters in a component in a model to not free.
<code>set_parameters_free([component_list, ...])</code>	Sets the parameters in a component in a model to free.
<code>set_parameters_value(parameter_name, value)</code>	Sets the value of a parameter in components in a model to a specified value
<code>as_signal([component_list, ...])</code>	Returns a recreation of the dataset using the model.
<code>export_results([folder, format, save_std, ...])</code>	Export the results of the parameters of the model to the desired folder.
<code>plot_results([only_free, only_active])</code>	Plot the value of the parameters of the model
<code>print_current_values([only_free, ...])</code>	Prints the current values of the parameters of all components.
<code>as_dictionary([fullcopy])</code>	Returns a dictionary of the model, including all components, degrees of freedom (dof) and chi-squared (chisq) with values.

property `active_components`

List all nonlinear parameters.

`append(thing)`

Add component to Model.

Parameters

`thing`

[`Component`] The component to add to the model.

`as_dictionary(fullcopy=True)`

Returns a dictionary of the model, including all components, degrees of freedom (dof) and chi-squared (chisq) with values.

Parameters

fullcopy

[**bool**, optional **True**] Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

Returns**dictionary**

[**dict**] A dictionary including at least the following fields:

- **components**: a list of dictionaries of components, one per component
- **_whitelist**: a dictionary with keys used as references for saved attributes, for more information, see [export_to_dictionary\(\)](#)
- any field from `_whitelist.keys()`

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> l1 = hs.model.components1D.Lorentzian()
>>> l2 = hs.model.components1D.Lorentzian()
>>> m.append(l1)
>>> m.append(l2)
>>> d = m.as_dictionary()
>>> m2 = s.create_model(dictionary=d)
```

as_signal(*component_list=None, out_of_range_to_nan=True, show_progressbar=None, out=None, **kwargs*)

Returns a recreation of the dataset using the model.

By default, the signal range outside of the fitted range is filled with nans.

Parameters**component_list**

[**list** of *Component*, optional] If a list of components is given, only the components given in the list is used in making the returned spectrum. The components can be specified by name, index or themselves.

out_of_range_to_nan

[**bool**] If **True** the signal range outside of the fitted range is filled with nans. Default **True**.

show_progressbar

[**None** or **bool**] If **True**, display a progress bar. If **None**, the default from the preferences settings is used.

out

[**None** or *BaseSignal*] The signal where to put the result into. Convenient for parallel processing. If **None** (default), creates a new one. If passed, it is assumed to be of correct shape and dtype and not checked.

Returns*BaseSignal*

The model as a signal.

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> l1 = hs.model.components1D.Lorentzian()
>>> l2 = hs.model.components1D.Lorentzian()
>>> m.append(l1)
>>> m.append(l2)
>>> s1 = m.as_signal()
>>> s2 = m.as_signal(component_list=[l1])
```

assign_current_values_to_all(*components_list=None, mask=None*)

Set parameter values for all positions to the current ones.

Parameters

component_list

[list of *Component*, optional] If a list of components is given, the operation will be performed only in the value of the parameters of the given components. The components can be specified by name, index or themselves. If None (default), the active components will be considered.

mask

[*numpy.ndarray* of *bool* or *None*, optional] The operation won't be performed where mask is True.

property **chisq**

Chi-squared of the signal (or np.nan if not yet fit).

property **components**

The components of the model are attributes of this class.

This provides a convenient way to access the model components when working in IPython as it enables tab completion.

create_samfire(*workers=None, setup=True, **kwargs*)

Creates a SAMFire object.

Parameters

workers

[*None* or *int*] the number of workers to initialise. If zero, all computations will be done serially. If None (default), will attempt to use (number-of-cores - 1), however if just one core is available, will use one worker.

setup

[*bool*] if the setup should be run upon initialization.

****kwargs**

Any that will be passed to the `_setup` and in turn `SamfirePool`.

disable_plot_components()

Disable interactive adjustment of the position of the components that have a well defined position. Use after `plot()`.

property **dof**

Degrees of freedom of the signal (0 if not yet fit)

enable_plot_components()

Enable interactive adjustment of the position of the components that have a well defined position. Use after `plot()`.

ensure_parameters_in_bounds()

For all active components, snaps their free parameter values to be within their boundaries (if bounded). Does not touch the array of values.

export_results(*folder=None, format='hspy', save_std=False, only_free=True, only_active=True*)

Export the results of the parameters of the model to the desired folder.

Parameters**folder**

[`str` or `None`] The path to the folder where the file will be saved. If `None` the current folder is used by default.

format

[`str`] The extension of the file format. It must be one of the fileformats supported by HyperSpy. The default is "hspy".

save_std

[`bool`] If True, also the standard deviation will be saved.

only_free

[`bool`] If True, only the value of the parameters that are free will be exported.

only_active

[`bool`] If True, only the value of the active parameters will be exported.

Notes

The name of the files will be determined by each the Component and each Parameter name attributes. Therefore, it is possible to customise the file names modify the name attributes.

extend(*iterable*)

Append multiple components to the model.

Parameters

iterable: iterable of ``Component`` instances.

fetch_stored_values(*only_fixed=False, update_on_resume=True*)

Fetch the value of the parameters that have been previously stored in `parameter.map['values']` if `parameter.map['is_set']` is `True` for those indices.

If it is not previously stored, the current values from `parameter.value` are used, which are typically from the fit in the previous pixel of a multidimensional signal.

Parameters**only_fixed**

[`bool`, optional] If True, only the fixed parameters are fetched.

update_on_resume

[`bool`, optional] If True, update the model plot after values are updated.

See also:

[store_current_values](#)

fetch_values_from_array(*array*, *array_std=None*)

Fetch the parameter values from the given array, optionally also fetching the standard deviations.

Places the parameter values into both *m.p0* (the initial values for the optimizer routine) and *component.parameter.value* and *...std*, for parameters in active components ordered by their position in the model and component.

Parameters

array

[[array](#)] array with the parameter values

array_std

[{[None](#), [array](#)}] array with the standard deviations of parameters

fit(*optimizer='lm'*, *loss_function='ls'*, *grad='fd'*, *bounded=False*, *update_plot=False*, *print_info=False*, *return_info=True*, *fd_scheme='2-point'*, ***kwargs*)

Fits the model to the experimental data.

Read more in the [User Guide](#).

Parameters

optimizer

[[str](#) or [None](#), default [None](#)] The optimization algorithm used to perform the fitting.

- "lm" performs least-squares optimization using the Levenberg-Marquardt algorithm, and supports bounds on parameters.
- "trf" performs least-squares optimization using the Trust Region Reflective algorithm, and supports bounds on parameters.
- "dogbox" performs least-squares optimization using the dogleg algorithm with rectangular trust regions, and supports bounds on parameters.
- "odr" performs the optimization using the orthogonal distance regression (ODR) algorithm. It does not support bounds on parameters. See [scipy.odr](#) for more details.
- All of the available methods for [scipy.optimize.minimize\(\)](#) can be used here. See the [User Guide](#) documentation for more details.
- "Differential Evolution" is a global optimization method. It does support bounds on parameters. See [scipy.optimize.differential_evolution\(\)](#) for more details on available options.
- "Dual Annealing" is a global optimization method. It does support bounds on parameters. See [scipy.optimize.dual_annealing\(\)](#) for more details on available options. Requires `scipy >= 1.2.0`.
- "SHGO" (simplicial homology global optimization) is a global optimization method. It does support bounds on parameters. See [scipy.optimize.shgo\(\)](#) for more details on available options. Requires `scipy >= 1.2.0`.

loss_function

[{"ls", "ML-poisson", "huber", [callable\(\)](#)}, default "ls"] The loss function to use for minimization. Only "ls" is available if *optimizer* is one of "lm", "trf", "dogbox" or "odr".

- "ls" minimizes the least-squares loss function.
- "ML-poisson" minimizes the negative log-likelihood for Poisson-distributed data. Also known as Poisson maximum likelihood estimation (MLE).

- "huber" minimize the Huber loss function. The delta value of the Huber function is controlled by the `huber_delta` keyword argument (the default value is 1.0).
- callable supports passing your own minimization function.

grad

[{"fd", "analytical", callable(), None}, default "fd"] Whether to use information about the gradient of the loss function as part of the optimization. This parameter has no effect if `optimizer` is a derivative-free or global optimization method.

- "fd" uses a finite difference scheme (if available) for numerical estimation of the gradient. The scheme can be further controlled with the `fd_scheme` keyword argument.
- "analytical" uses the analytical gradient (if available) to speed up the optimization, since the gradient does not need to be estimated.
- callable should be a function that returns the gradient vector.
- None means that no gradient information is used or estimated. Not available if `optimizer` is one of "lm", "trf" or "dogbox".

bounded

[bool, default False] If True, performs bounded parameter optimization if supported by `optimizer`.

update_plot

[bool, default False] If True, the plot is updated during the optimization process. It slows down the optimization, but it enables visualization of the optimization progress.

print_info

[bool, default False] If True, print information about the fitting results, which are also stored in `model.fit_output` in the form of a `scipy.optimize.OptimizeResult` object.

return_info

[bool, default True] If True, returns the fitting results in the form of a `scipy.optimize.OptimizeResult` object.

fd_scheme

[str {"2-point", "3-point", "cs"}, default "2-point"] If `grad='fd'`, selects the finite difference scheme to use. See `scipy.optimize.minimize()` for details. Ignored if `optimizer` is one of "lm", "trf" or "dogbox".

****kwargs**

[dict] Any extra keyword argument will be passed to the chosen optimizer. For more information, read the docstring of the optimizer of your choice in `scipy.optimize`.

Returns

None

See also:

multifit, *fit*

Notes

The chi-squared and reduced chi-squared statistics, and the degrees of freedom, are computed automatically when fitting, only when `loss_function="ls"`. They are stored as signals: `chisq`, `red_chisq` and `dof`.

If the attribute `metada.Signal.Noise_properties.variance` is defined as a `Signal` instance with the same `navigation_dimension` as the signal, and `loss_function` is "ls" or "huber", then a weighted fit is performed, using the inverse of the noise variance as the weights.

Note that for both homoscedastic and heteroscedastic noise, if `metadata.Signal.Noise_properties.variance` does not contain an accurate estimation of the variance of the data, then the chi-squared and reduced chi-squared statistics will not be computed correctly. See the *Setting the noise properties* in the User Guide for more details.

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

insert (***kwargs*)

Insert object before index.

load_parameters_from_file (*filename*)

Loads the parameters array from a binary file written with the `save_parameters2file()` function.

Parameters

filename

[str] The file name of the file to load it from.

See also:

[save_parameters2file](#), [export_results](#)

Notes

In combination with `save_parameters2file()`, this method can be used to recreate a model stored in a file. Actually, before HyperSpy 0.8 this is the only way to do so. However, this is known to be brittle. For example see <https://github.com/hyperspy/hyperspy/issues/341>.

multifit (*mask=None, fetch_only_fixed=False, autosave=False, autosave_every=10, show_progressbar=None, interactive_plot=False, iterpath=None, **kwargs*)

Fit the data to the model at all positions of the navigation dimensions.

Parameters

mask

[np.ndarray, optional] To mask (i.e. do not fit) at certain position, pass a boolean numpy.array, where True indicates that the data will NOT be fitted at the given position.

fetch_only_fixed

[[bool](#), default [False](#)] If True, only the fixed parameters values will be updated when changing the position.

autosave

[[bool](#), default [False](#)] If True, the result of the fit will be saved automatically with a frequency defined by `autosave_every`.

autosave_every

[[int](#), default 10] Save the result of fitting every given number of spectra.

show_progressbar

[[None](#) or [bool](#)] If True, display a progress bar. If [None](#), the default from the preferences settings is used.

interactive_plot

[[bool](#), default [False](#)] If True, update the plot for every position as they are processed. Note that this slows down the fitting by a lot, but it allows for interactive monitoring of the fitting (if in interactive mode).

iterpath

[[[None](#), "flyback", "serpentine"], default [None](#)]

If "flyback":

At each new row the index begins at the first column, in accordance with the way `numpy.ndindex` generates indices.

If "serpentine":

Iterate through the signal in a serpentine, “snake-game”-like manner instead of beginning each new row at the first index. Works for n-dimensional navigation space, not just 2D.

If None:

Use the value of `iterpath`.

****kwargs**

[[dict](#)] Any extra keyword argument will be passed to the fit method. See the documentation for `fit()` for a list of valid arguments.

Returns

[None](#)

See also:

[fit](#)

`plot_results(only_free=True, only_active=True)`

Plot the value of the parameters of the model

Parameters**only_free**

[[bool](#)] If True, only the value of the parameters that are free will be plotted.

only_active

[[bool](#)] If True, only the value of the active parameters will be plotted.

Notes

The name of the files will be determined by each the Component and each Parameter name attributes. Therefore, it is possible to customise the file names modify the name attributes.

print_current_values(*only_free=False, only_active=False, component_list=None*)

Prints the current values of the parameters of all components.

Parameters

only_free

[bool] If True, only components with free parameters will be printed. Within these, only parameters which are free will be printed.

only_active

[bool] If True, only values of active components will be printed

component_list

[None or list of *Component*] If None, print all components.

property red_chisq

The Reduced chi-squared.

Calculated from `self.chisq` and `self.dof`.

remove(*thing*)

Remove component from model.

Examples

```
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1, g2])
```

You could remove g1 like this

```
>>> m.remove(g1)
```

Or like this:

```
>>> m.remove(0)
```

save(*file_name, name=None, **kwargs*)

Saves signal and its model to a file

Parameters

file_name

[str] Name of the file

name

[{None, str}] Stored model name. Auto-generated if left empty

****kwargs**

Other keyword arguments are passed onto *BaseSignal.save()*

save_parameters2file(filename)

Save the parameters array in binary format.

The data is saved to a single file in numpy's uncompressed .npz format.

Parameters**filename**

[str] The file name of the file it is saved to.

See also:

[load_parameters_from_file](#), [export_results](#)

Notes

This method can be used to save the current state of the model in a way that can be loaded back to recreate it using [load_parameters_from_file\(\)](#). Actually, as of HyperSpy 0.8 this is the only way to do so. However, this is known to be brittle. For example see <https://github.com/hyperspy/hyperspy/issues/341>.

set_component_active_value(value, component_list=None, only_current=False)

Sets the component 'active' parameter to a specified value.

Parameters**value**

[bool] The new value of the 'active' parameter

component_list

[list of [Component](#), optional] A list of components whose parameters will be changed. The components can be specified by name, index or themselves.

only_current

[bool, default `False`] If `True`, will only change the parameter value at the current position in the model. If `False`, will change the parameter value for all the positions.

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> v1 = hs.model.components1D.Voigt()
>>> v2 = hs.model.components1D.Voigt()
>>> m.extend([v1,v2])
```

```
>>> m.set_component_active_value(False)
>>> m.set_component_active_value(True, component_list=[v1])
>>> m.set_component_active_value(
...     False, component_list=[v1], only_current=True
... )
```

set_parameters_free(component_list=None, parameter_name_list=None, only_linear=False, only_nonlinear=False)

Sets the parameters in a component in a model to free.

Parameters

component_list

[None or list of *Component*, optional] If None, will apply the function to all components in the model. If list of components, will apply the functions to the components in the list. The components can be specified by name, index or themselves.

parameter_name_list

[None or list of *str*, optional] If None, will set all the parameters to not free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to not free.

only_linear

[bool] If True, will only set parameters that are linear to not free.

only_nonlinear

[bool] If True, will only set parameters that are nonlinear to not free.

See also:

set_parameters_not_free

hyperspy.component.Component.set_parameters_free

hyperspy.component.Component.set_parameters_not_free

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> v1 = hs.model.components1D.Voigt()
>>> m.append(v1)
```

```
>>> m.set_parameters_free()
>>> m.set_parameters_free(
...     component_list=[v1], parameter_name_list=['area','centre']
... )
>>> m.set_parameters_free(only_linear=True)
```

set_parameters_not_free(*component_list=None, parameter_name_list=None, only_linear=False, only_nonlinear=False*)

Sets the parameters in a component in a model to not free.

Parameters**component_list**

[None or list of *Component*, optional] If None, will apply the function to all components in the model. If list of components, will apply the functions to the components in the list. The components can be specified by name, index or themselves.

parameter_name_list

[None or list of *str*, optional] If None, will set all the parameters to not free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to not free.

only_linear

[bool] If True, will only set parameters that are linear to free.

only_nonlinear

[bool] If True, will only set parameters that are nonlinear to free.

See also:

[`set_parameters_free`](#)

[`hyperspy.component.Component.set_parameters_free`](#)

[`hyperspy.component.Component.set_parameters_not_free`](#)

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> v1 = hs.model.components1D.Voigt()
>>> m.append(v1)
>>> m.set_parameters_not_free()
```

```
>>> m.set_parameters_not_free(
...     component_list=[v1], parameter_name_list=['area','centre']
... )
>>> m.set_parameters_not_free(only_linear=True)
```

set_parameters_value(*parameter_name*, *value*, *component_list=None*, *only_current=False*)

Sets the value of a parameter in components in a model to a specified value

Parameters

parameter_name

[*str*] Name of the parameter whose value will be changed

value

[*float* or *int*] The new value of the parameter

component_list

[*None* or *list* of *Component*, optional] A list of components whose parameters will be changed. The components can be specified by name, index or themselves. If *None*, use all components of the model.

only_current

[*bool*, default *False*] If *True*, will only change the parameter value at the current position in the model. If *False*, will change the parameter value for all the positions.

Examples

```
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> v1 = hs.model.components1D.Voigt()
>>> v2 = hs.model.components1D.Voigt()
>>> m.extend([v1,v2])
```

```
>>> m.set_parameters_value('area', 5)
>>> m.set_parameters_value('area', 5, component_list=[v1])
>>> m.set_parameters_value(
...     'area', 5, component_list=[v1], only_current=True
... )
```

set_signal_range_from_mask(*mask*)

Use the signal ranges as defined by the mask

Parameters

mask

[[numpy.ndarray](#) of [bool](#)] A boolean array defining the signal range. Must be the same shape as the reversed `signal_shape`, i.e. `signal_shape[::-1]`. Where array values are True, signal will be fitted, otherwise not.

See also:

[hyperspy.models.model1d.Model1D.set_signal_range](#)
[hyperspy.models.model1d.Model1D.add_signal_range](#)
[hyperspy.models.model1d.Model1D.remove_signal_range](#)
[hyperspy.models.model1d.Model1D.reset_signal_range](#)

Examples

```
>>> s = hs.signals.Signal2D(np.random.rand(10, 10, 20))
>>> mask = (s.sum() > 5)
>>> m = s.create_model()
>>> m.set_signal_range_from_mask(mask.data)
```

property signal

The signal data to fit.

store(*name=None*)

Stores current model in the original signal

Parameters

name

[[None](#), [str](#)] Stored model name. Auto-generated if left empty

store_current_values()

Store the parameters of the current coordinates into the *parameter.map* array and sets the *is_set* array attribute to True.

If the parameters array has not being defined yet it creates it filling it with the current parameters at the current indices in the array.

suspend_update(*update_on_resume=True*)

Prevents plot from updating until 'with' clause completes.

See also:

[update_plot](#)

update_plot(*render_figure=False, update_ylimits=False, **kwargs*)

Update model plot.

The updating can be suspended using *suspend_update*.

See also:

[suspend_update](#)

```
class hyperspy.model.ModelComponents(model)
```

Bases: `object`

Container for model components.

Useful to provide tab completion when running in IPython.

Model1D

```
class hyperspy.models.model1d.Model1D(signal1D, dictionary=None)
```

Bases: `BaseModel`

Model and data fitting for one dimensional signals.

A model is constructed as a linear combination of `components1D` that are added to the model using `append()` or `extend()`. There are many predefined components available in the `components1D` module. If needed, new components can be created easily using the `Expression` component or by using the code of existing components as a template.

Once defined, the model can be fitted to the data using `fit()` or `multifit()`. Once the optimizer reaches the convergence criteria or the maximum number of iterations the new value of the component parameters are stored in the components.

It is possible to access the components in the model by their name or by the index in the model. An example is given at the end of this docstring.

See also:

[hyperspy.model.BaseModel](#), [hyperspy.models.model2d.Model2D](#)

Examples

In the following example we create a histogram from a normal distribution and fit it with a gaussian component. It demonstrates how to create a model from a `Signal1D` instance, add components to it, adjust the value of the parameters of the components, fit the model to the data and access the components in the model.

```
>>> s = hs.signals.Signal1D(
...     np.random.normal(scale=2, size=10000)).get_histogram()
>>> g = hs.model.components1D.Gaussian()
>>> m = s.create_model()
>>> m.append(g)
>>> m.print_current_values()
Model1D: histogram
CurrentComponentValues: Gaussian
Active: True
Parameter Name | Free | Value | Std | Min | Max |
<-Linear
===== | ===== | ===== | ===== | ===== | ===== |
<-=====
           A | True | 1.0 | None | 0.0 | None |
<-True
       centre | True | 0.0 | None | None | None |
<-False
         sigma | True | 1.0 | None | 0.0 | None |
<-False
```

(continues on next page)

(continued from previous page)

```

>>> g.centre.value = 3
>>> m.print_current_values()
ModelID: histogram
CurrentComponentValues: Gaussian
Active: True
Parameter Name | Free | Value | Std | Min | Max |
↳Linear
===== | ===== | ===== | ===== | ===== | ===== |
↳=====
           A | True | 1.0 | None | 0.0 | None |
↳True
       centre | True | 3.0 | None | None | None |
↳False
           sigma | True | 1.0 | None | 0.0 | None |
↳False
>>> g.sigma.value
1.0
>>> m.fit()
>>> g.sigma.value
1.9779042300856682
>>> m[0].sigma.value
1.9779042300856682
>>> m["Gaussian"].centre.value
-0.072121936813224569

```

Methods

<code>fit_component(component[, signal_range, ...])</code>	Fit the given component in the given signal range.
<code>enable_adjust_position([components, ...])</code>	Allow changing the x position of component by dragging a vertical line that is plotted in the signal model figure
<code>disable_adjust_position()</code>	Disable the interactive adjust position feature
<code>plot([plot_components, plot_residual])</code>	Plot the current spectrum to the screen and a map with a cursor to explore the SI.
<code>set_signal_range(*args, **kwargs)</code>	Use only the selected spectral range defined in its own units in the fitting routine.
<code>remove_signal_range(*args, **kwargs)</code>	Removes the data in the given range from the data range that will be used by the fitting routine.
<code>reset_signal_range()</code>	Resets the data range
<code>add_signal_range(*args, **kwargs)</code>	Adds the data in the given range from the data range that will be used by the fitting routine.

`add_signal_range(*args, **kwargs)`

Adds the data in the given range from the data range that will be used by the fitting routine.

Parameters

x1, x2
[None or float]

See also:

set_signal_range, reset_signal_range, remove_signal_range

append(*thing*)

Add component to Model.

Parameters

thing

[*Component*] The component to add to the model.

disable_adjust_position()

Disable the interactive adjust position feature

See also:

enable_adjust_position

disable_plot_components()

Disable interactive adjustment of the position of the components that have a well defined position. Use after *plot()*.

enable_adjust_position(*components=None, fix_them=True, show_label=True*)

Allow changing the *x* position of component by dragging a vertical line that is plotted in the signal model figure

Parameters

components

[*None, list of Component*] If *None*, the position of all the active components of the model that has a well defined *x* position with a value in the axis range will get a position adjustment line. Otherwise the feature is added only to the given components. The components can be specified by name, index or themselves.

fix_them

[*bool, default True*] If *True* the position parameter of the components will be temporarily fixed until adjust position is disable. This can be useful to iteratively adjust the component positions and fit the model.

show_label

[*bool, default True*] If *True*, a label showing the component name is added to the plot next to the vertical line.

See also:

disable_adjust_position

enable_plot_components()

Enable interactive adjustment of the position of the components that have a well defined position. Use after *plot()*.

fit_component(*component, signal_range='interactive', estimate_parameters=True, fit_independent=False, only_current=True, display=True, toolkit=None, **kwargs*)

Fit the given component in the given signal range.

This method is useful to obtain starting parameters for the components. Any keyword arguments are passed to the fit method.

Parameters

component

[*Component*] The component must be in the model, otherwise an exception is raised. The component can be specified by name, index or itself.

signal_range

[*str*, *tuple* of *None*] If 'interactive' the signal range is selected using the span selector on the spectrum plot. The signal range can also be manually specified by passing a tuple of floats (left, right). If *None* the current signal range is used. Note that ROIs can be used in place of a tuple.

estimate_parameters

[*bool*, default *True*] If *True* will check if the component has an `estimate_parameters` function, and use it to estimate the parameters in the component.

fit_independent

[*bool*, default *False*] If *True*, all other components are disabled. If *False*, all other component parameters are fixed.

display

[*bool*] If *True*, display the user interface widgets. If *False*, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[*str*, *iterable* of *str* or *None*] If *None* (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an iterable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

****kwargs**

[*dict*] All extra keyword arguments are passed to the `py:meth:~hyperspy.model.BaseModel.fit` or `py:meth:~hyperspy.model.BaseModel.multifit` method, depending if `only_current` is *True* or *False*.

Examples

Signal range set interactively

```
>>> s = hs.signals.Signal1D([0, 1, 2, 4, 8, 4, 2, 1, 0])
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> m.append(g1)
>>> m.fit_component(g1)
```

Signal range set through direct input

```
>>> m.fit_component(g1, signal_range=(1, 7))
```

plot(*plot_components=False*, *plot_residual=False*, ***kwargs*)

Plot the current spectrum to the screen and a map with a cursor to explore the SI.

Parameters**plot_components**

[*bool*] If *True*, add a line per component to the signal figure.

plot_residual

[*bool*] If *True*, add a residual line (Signal - Model) to the signal figure.

****kwargs**
[dict] All extra keyword arguments are passed to `plot()`

remove(*things*)

Remove component from model.

Examples

```
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1, g2])
```

You could remove `g1` like this

```
>>> m.remove(g1)
```

Or like this:

```
>>> m.remove(0)
```

remove_signal_range(*args, **kwargs)

Removes the data in the given range from the data range that will be used by the fitting routine.

Parameters

x1, x2
[None or float]

See also:

[*set_signal_range*](#), [*add_signal_range*](#), [*reset_signal_range*](#)
[*hyperspy.model.BaseModel.set_signal_range_from_mask*](#)

reset_signal_range()

Resets the data range

See also:

[*set_signal_range*](#), [*add_signal_range*](#), [*remove_signal_range*](#)

set_signal_range(*args, **kwargs)

Use only the selected spectral range defined in its own units in the fitting routine.

Parameters

x1, x2
[None or float]

See also:

[*add_signal_range*](#), [*remove_signal_range*](#), [*reset_signal_range*](#)
[*hyperspy.model.BaseModel.set_signal_range_from_mask*](#)

Model2D

class `hyperspy.models.model2d.Model2D`(*signal2D*, *dictionary=None*)

Bases: *BaseModel*

Model and data fitting for two dimensional signals.

A model is constructed as a linear combination of *components2D* that are added to the model using *append()* or *extend()*. There are predefined components available in the *components2D* module and custom components can be made using the *Expression*. If needed, new components can be created easily using the code of existing components as a template.

Once defined, the model can be fitted to the data using *fit()* or *multifit()*. Once the optimizer reaches the convergence criteria or the maximum number of iterations the new value of the component parameters are stored in the components.

It is possible to access the components in the model by their name or by the index in the model. An example is given at the end of this docstring.

See also:

hyperspy.model.BaseModel, *hyperspy.models.model1d.Model1D*

Notes

Methods are not yet defined for plotting 2D models or using gradient based optimisation methods.

Methods

<code>add_signal_range([x1, x2, y1, y2])</code>	Adds the data in the given range from the data range (calibrated values) that will be used by the fitting routine.
<code>remove_signal_range([x1, x2, y1, y2])</code>	Removes the data in the given range (calibrated values) from the data range that will be used by the fitting routine
<code>reset_signal_range()</code>	Resets the data range.
<code>set_signal_range([x1, x2, y1, y2])</code>	Use only the selected range defined in its own units in the fitting routine.

add_signal_range(*x1=None*, *x2=None*, *y1=None*, *y2=None*)

Adds the data in the given range from the data range (calibrated values) that will be used by the fitting routine.

Parameters

x1, x2

[None or float] Start and end of the range in the first axis (horizontal) in units.

y1, y2

[None or float] Start and end of the range in the second axis (vertical) in units.

See also:

set_signal_range, *reset_signal_range*, *remove_signal_range*
hyperspy.model.BaseModel.set_signal_range_from_mask

remove_signal_range(*x1=None, x2=None, y1=None, y2=None*)

Removes the data in the given range (calibrated values) from the data range that will be used by the fitting routine

Parameters

x1, x2

[None or float] Start and end of the range in the first axis (horizontal) in units.

y1, y2

[None or float] Start and end of the range in the second axis (vertical) in units.

See also:

set_signal_range, add_signal_range, reset_signal_range
hyperspy.model.BaseModel.set_signal_range_from_mask

reset_signal_range()

Resets the data range.

See also:

set_signal_range, add_signal_range, remove_signal_range
hyperspy.model.BaseModel.set_signal_range_from_mask

set_signal_range(*x1=None, x2=None, y1=None, y2=None*)

Use only the selected range defined in its own units in the fitting routine.

Parameters

x1, x2

[None or float] Start and end of the range in the first axis (horizontal) in units.

y1, y2

[None or float] Start and end of the range in the second axis (vertical) in units.

See also:

add_signal_range, remove_signal_range, reset_signal_range
hyperspy.model.BaseModel.set_signal_range_from_mask

Component

class `hyperspy.component.Component`(*parameter_name_list, linear_parameter_list=None, *args, **kwargs*)

Bases: `HasTraits`

The component of a model.

Attributes

active

[bool]

name

[str]

free_parameters

[list] The list of free parameters of the component.

parameters

[list] The list of parameters of the component.

Parameters**parameter_name_list**

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

property active_is_multidimensional

In multidimensional signals it is possible to store the value of the active attribute at each navigation index.

as_dictionary(*fullcopy=True*)

Returns component as a dictionary. For more information on method and conventions, see [export_to_dictionary\(\)](#).

Parameters**fullcopy**

[bool, optional `False`] Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

Returns**dic**

[dict] A dictionary, containing at least the following fields:

- parameters: a list of dictionaries of the parameters, one per component.
- _whitelist: a dictionary with keys used as references saved attributes, for more information, see [export_to_dictionary\(\)](#).
- any field from `_whitelist.keys()`.

export(*folder=None, format='hspy', save_std=False, only_free=True*)

Plot the value of the parameters of the model

Parameters**folder**

[str or None] The path to the folder where the file will be saved. If `None` the current folder is used by default.

format

[str] The extension of the file format, default “hspy”.

save_std

[bool] If True, also the standard deviation will be saved.

only_free

[bool] If True, only the value of the parameters that are free will be exported.

Notes

The name of the files will be determined by each the Component and each Parameter name attributes. Therefore, it is possible to customise the file names modify the name attributes.

fetch_values_from_array(*p*, *p_std=None*, *onlyfree=False*)

Fetch the parameter values from an array *p* and optionally standard deviation from *p_std*. Places them *component.parameter.value* and *...std*, according to their position in the component.

Parameters

p

[array] array containing new values for the parameters in a component

p_std

[array, optional] array containing the corresponding standard deviation.

property free_parameters

The list of free parameters of the component.

gui(*display=True*, *toolkit=None*, ***kwargs*)

Display or return interactive GUI element if available.

Parameters

display

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

init_parameters(*parameter_name_list*, *linear_parameter_list=None*)

Initialise the parameters of the component.

Parameters

parameter_name_list

[list] The list of parameter names.

linear_parameter_list

[list, optional] The list of linear parameter. The default is None.

property parameters

The list of parameters of the component.

plot(*only_free=True*)

Plot the value of the parameters of the model

Parameters

only_free

[bool]

If True, only the value of the parameters that are free will be plotted

print_current_values(*only_free=False*)

Prints the current values of the component's parameters.

Parameters**only_free**

[bool] If True, only free parameters will be printed.

set_parameters_free(*parameter_name_list=None, only_linear=False, only_nonlinear=False*)

Sets parameters in a component to free.

Parameters**parameter_name_list**

[None or list of str, optional] If None, will set all the parameters to free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to free.

only_linear

[bool] If True, only sets a parameter free if it is linear

only_nonlinear

[bool] If True, only sets a parameter free if it is nonlinear

See also:

[*set_parameters_not_free*](#)

[*hyperspy.model.BaseModel.set_parameters_free*](#)

[*hyperspy.model.BaseModel.set_parameters_not_free*](#)

Examples

```
>>> v1 = hs.model.components1D.Voigt()
>>> v1.set_parameters_free()
>>> v1.set_parameters_free(parameter_name_list=['area', 'centre'])
>>> v1.set_parameters_free(only_linear=True)
```

set_parameters_not_free(*parameter_name_list=None, only_linear=False, only_nonlinear=False*)

Sets parameters in a component to not free.

Parameters**parameter_name_list**

[None or list of str, optional] If None, will set all the parameters to not free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to not free.

only_linear

[bool] If True, only sets a parameter not free if it is linear

only_nonlinear

[bool] If True, only sets a parameter not free if it is nonlinear

See also:

[*set_parameters_free*](#)

[*hyperspy.model.BaseModel.set_parameters_free*](#)

[*hyperspy.model.BaseModel.set_parameters_not_free*](#)

Examples

```
>>> v1 = hs.model.components1D.Voigt()
>>> v1.set_parameters_not_free()
>>> v1.set_parameters_not_free(parameter_name_list=['area', 'centre'])
>>> v1.set_parameters_not_free(only_linear=True)
```

Parameter

class hyperspy.component.**Parameter**

Bases: `HasTraits`

The parameter of a component.

Attributes

bmin

[float] The lower bound of the parameter.

bmax

[float] The upper bound of the parameter.

ext_force_positive

[bool] If True, the parameter value is set to be the absolute value of the input value i.e.

ext_bounded

[bool] Similar to *ext_force_positive*, but in this case the bounds are defined by *bmin* and *bmax*.

free

[bool]

map

[numpy.ndarray]

twin

[None or *Parameter*] If it is not None, the value of the current parameter is a function of the given *Parameter*.

twin_function_expr

[str] Expression of the function that enables setting a functional relationship between the parameter and its twin.

twin_inverse_function_expr

[str] Expression of the function that enables setting the value of the twin parameter.

value

[float or array] The value of the parameter for the current location. The value for other locations is stored in *map*.

value

free

map

as_dictionary(*fullcopy=True*)

Returns parameter as a dictionary, saving all attributes from `self._whitelist.keys()` For more information see `py:meth:~hyperspy.misc.export_dictionary.export_to_dictionary`

Parameters**fullcopy**

[`bool`, optional `False`] Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

Returns**dict**

A dictionary, containing at least the following fields:

- `_id_name`: `_id_name` of the original parameter, used to create the dictionary. Has to match with the `self._id_name`
- `_twins`: a list of ids of the twins of the parameter
- `_whitelist`: a dictionary, which keys are used as keywords to match with the parameter attributes. For more information see [export_to_dictionary\(\)](#)
- any field from `_whitelist.keys()`

as_signal(*field='values'*)

Get a parameter map as a signal object.

Please note that this method only works when the navigation dimension is greater than 0.

Parameters**field**

[{'values', 'std', 'is_set'}] Field to return as signal.

assign_current_value_to_all(*mask=None*)

Assign the current value attribute to all the indices, setting `parameter.map` for all parameters in the component.

Takes the current `parameter.value` and sets it for all indices in `parameter.map['values']`.

Parameters**mask: {None, boolean numpy array}**

Set only the indices that are not masked i.e. where `mask` is `False`.

See also:

[store_current_value_in_array](#), [fetch](#)

default_traits_view()

Returns the name of the default traits view for the object's class.

export(*folder=None, name=None, format='hspy', save_std=False*)

Save the data to a file. All the arguments are optional.

Parameters**folder**

[`str` or `None`] The path to the folder where the file will be saved. If `None` the current folder is used by default.

name

[`str` or `None`] The name of the file. If `None` the Components name followed by the Parameter `name` attributes will be used by default. If a file with the same name exists the name will be modified by appending a number to the file path.

save_std

[bool] If True, also the standard deviation will be saved

format: str

The extension of any file format supported by HyperSpy, default hspy.

property ext_bounded

Similar to *ext_force_positive*, but in this case the bounds are defined by *bmin* and *bmax*. It is a better idea to use an optimizer that supports bounding though.

property ext_force_positive

If True, the parameter value is set to be the absolute value of the input value i.e. if we set `Parameter.value = -3`, the value stored is 3 instead. This is useful to bound a value to be positive in an optimization without actually using an optimizer that supports bounding.

fetch()

Fetch the stored value and std attributes from the `parameter.map['values']` and `...['std']` if `parameter.map['is_set']` is True for that index. Updates `parameter.value` and `parameter.std`. If not stored, then `.value` and `.std` will remain from their previous values, i.e. from a fit in a previous pixel.

See also:

[*store_current_value_in_array*](#), [*assign_current_value_to_all*](#)

gui (*display=True, toolkit=None, **kwargs*)

Display or return interactive GUI element if available.

Parameters**display**

[bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

toolkit

[str, iterable of str or None] If None (default), all available widgets are displayed or returned. If string, only the widgets of the selected toolkit are displayed if available. If an interable of toolkit strings, the widgets of all listed toolkits are displayed or returned.

plot (***kwargs*)

Plot parameter signal.

Parameters****kwargs**

Any extra keyword arguments are passed to the signal plot.

Examples

```
>>> parameter.plot()
```

Set the minimum and maximum displayed values

```
>>> parameter.plot(vmin=0, vmax=1)
```

store_current_value_in_array()

Store the value and std attributes.

See also:

fetch, assign_current_value_to_all

property twin

If it is not None, the value of the current parameter is a function of the given Parameter. The function is by default the identity function, but it can be defined by *twin_function_expr*

property twin_function_expr

Expression of the function that enables setting a functional relationship between the parameter and its twin. If *twin* is not None, the parameter value is calculated as the output of calling the twin function with the value of the twin parameter. The string is parsed using sympy, so permitted values are any valid sympy expressions of one variable. If the function is invertible the twin inverse function is set automatically.

property twin_inverse_function_expr

Expression of the function that enables setting the value of the twin parameter. If *twin* is not None, its value is set to the output of calling the twin inverse function with the value provided. The string is parsed using sympy, so permitted values are any valid sympy expressions of one variable.

SamFire

class `hyperspy.samfire.Samfire`(*model, workers=None, setup=True, random_state=None, **kwargs*)

Bases: `object`

Smart Adaptive Multidimensional Fitting (SAMFire) object

SAMFire is a more robust way of fitting multidimensional datasets. By extracting starting values for each pixel from already fitted pixels, SAMFire stops the fitting algorithm from getting lost in the parameter space by always starting close to the optimal solution.

SAMFire only picks starting parameters and the order the pixels (in the navigation space) are fitted, and does not provide any new minimisation algorithms.

Attributes

model

[*hyperspy.model.BaseModel* (or subclass)] The complete model

optional_components

[*list*] A list of components that can be switched off at some pixels if it returns a better Akaike's Information Criterion with correction (AICc)

workers

[*int*] A number of processes that will perform the fitting parallelly

pool

[*SamfirePool*] A proxy object that manages either multiprocessing or ipyparallel pool

strategies

[*list*] A list of strategies that will be used to select pixel fitting order and calculate required starting parameters. Strategies come in two "flavours" - local and global. Local strategies spread the starting values to the nearest pixels and forces certain pixel fitting order. Global strategies look for clusters in parameter values, and suggests most frequent values. Global strategy do not depend on pixel fitting order, hence it is randomised.

metadata

[*dict*] A dictionary for important samfire parameters

active_strategy

[*strategy*] The currently active strategy from the strategies list

update_every

[[int](#)] If segmenter strategy is running, updates the historams every time `update_every` good fits are found.

plot_every

[[int](#)] When running, samfire plots results every time `plot_every` good fits are found.

save_every

[[int](#)] When running, samfire saves results every time `save_every` good fits are found.

random_state

[[None](#) or [int](#) or `numpy.random.Generator`, default [None](#)] Random seed used to select the next pixels.

append(*strategy*)

Append the given strategy to the end of the strategies list

Parameters**strategy**

[[strategy](#)] The samfire strategy to use

backup(*filename=None, on_count=True*)

Backup the samfire results in a file.

Parameters**filename**

[[str](#), [None](#), default [None](#)] the filename. If [None](#), a default value of `backup_ + signal_title` is used.

on_count

[[bool](#), default [True](#)] if [True](#), only saves on the required count of steps

change_strategy(*new_strat*)

Changes current strategy to a new one. Certain rules apply: diffusion -> diffusion : resets all “ignored” pixels diffusion -> segmenter : saves already calculated pixels to be ignored when(if) subsequently diffusion strategy is run

Parameters**new_strat**

[[int](#) or [strategy](#)] index of the new strategy from the strategies list or the strategy object itself

extend(*iterable*)

Extend the strategies list by the given iterable

Parameters**iterable**

[[iterable](#) of [strategy](#)] The samfire strategies to use.

generate_values(*need_inds*)

Returns an iterator that yields the index of the pixel and the value dictionary to be sent to the workers.

Parameters**need_inds**

[[int](#)] the number of pixels to be returned in the generator

log(*args)

If has a list named “_log” as attribute, appends the arguments there

property pixels_done

Returns the number of pixels that have been solved

property pixels_left

Returns the number of pixels that are left to solve. This number can increase as SAMFire learns more information about the data.

plot(on_count=False)

If possible, plot current strategy plot. Local strategies plot grayscale navigation signal with brightness representing order of the pixel selection. Global strategies plot a collection of histograms, one per parameter.

Parameters

on_count

[bool] if True, only tries to plot every specified count, otherwise (default) always plots if possible.

refresh_database()

Refresh currently selected strategy without preserving any “ignored” pixels; no previous structure is preserved.

remove(thing)

Remove given strategy from the strategies list

Parameters

thing

[int or strategy] Strategy that is in current strategies list or its index.

start(**kwargs)

Start SAMFire.

Parameters

****kwargs**

[dict] Any keyword arguments to be passed to *fit*()

stop()

Stop SAMFire.

update(ind, results=None, isgood=None)

Updates the current model with the results, received from the workers. Results are only stored if the results are good enough

Parameters

ind

[tuple] contains the index of the pixel of the results

results

[dict or None, default None] dictionary of the results. If None, means we are updating in-place (e.g. refreshing the marker or strategies).

isgood

[bool or None, default None] if it is known if the results are good according to the goodness-of-fit test. If None, the pixel is tested.

```
class hyperspy.utils.parallel_pool.ParallelPool(num_workers=None, ipython_kwargs=None,
                                                ipyparallel=None)
```

Bases: `object`

Creates a ParallelPool by either looking for a ipyparallel client and then creating a load_balanced_view, or by creating a multiprocessing pool

Attributes

pool

[`ipyparallel.LoadBalancedView` or `multiprocessing.pool.Pool`] The pool object.

ipython_kwargs

[`dict`] The dictionary with Ipyparallel connection arguments.

timeout

[`float`] Timeout for either pool when waiting for results.

num_workers

[`int`] The number of workers actually created (may be less than requested, but can't be more).

timestep

[`float`] Can be used as “ticks” to adjust CPU load when building upon this class.

Creates the ParallelPool and sets it up.

Parameters

num_workers

[`None` or `int`, default `None`] The (max) number of workers to create. If less are available, smaller number is actually created.

ipyparallel

[`None` or `bool`, default `None`] Which pool to set up. True - ipyparallel. False - multiprocessing. None - try ipyparallel, then multiprocessing if failed.

ipython_kwargs

[`None` or `dict`, default `None`] Arguments that will be passed to the ipyparallel.Client when creating. Not None implies ipyparallel=True.

property has_pool

Returns True if the pool is ready and set-up else False.

property is_ipyparallel

Returns True if the pool is ipyparallel-based else False.

property is_multiprocessing

Returns True if the pool is multiprocessing-based else False.

setup(ipyparallel=None)

Sets up the pool.

Parameters

ipyparallel

[`None` or `bool`, default `None`] If True, only tries to set up the ipyparallel pool. If False - only the multiprocessing. If None, first tries ipyparallel, and it does not succeed, then multiprocessing.

sleep(*howlong=None*)

Sleeps for the required number of seconds.

Parameters

howlong

[None or float] How long the pool should sleep for in seconds. If None (default), sleeps for “timestep”.

23.9.5 Signal

API of signal classes, which are not part of the user-facing *hyperspy.api* namespace but are inherited in HyperSpy signals classes or used as attributes of signals. These classes are not expected to be instantiated by users but their methods, which are used by other classes, are documented here.

ModelManager

<i>ModelManager</i>	Container for models
---------------------	----------------------

Common Signals

<i>CommonSignal1D</i>	Common functions for 1-dimensional signals.
-----------------------	---

<i>CommonSignal2D</i>	Common functions for 2-dimensional signals.
-----------------------	---

Lazy Signals

<i>LazySignal</i>	Lazy general signal class.
-------------------	----------------------------

<i>LazyComplexSignal</i>	Lazy general signal class for complex data.
<i>LazyComplexSignal1D</i>	Lazy signal class for complex 1-dimensional data.
<i>LazyComplexSignal2D</i>	Lazy Signal class for complex 2-dimensional data.
<i>LazySignal1D</i>	Lazy general 1D signal class.
<i>LazySignal2D</i>	Lazy general 2D signal class.

ModelManager

class hyperspy.signal.**ModelManager**(*signal*, *dictionary=None*)

Bases: `object`

Container for models

pop(*name*)

Returns the restored model and removes it from storage

Parameters

name

[`str`] The name of the model to restore and remove

See also:

`restore`

`store`

`remove`

remove(*name*)

Removes the given model

Parameters

name

[`str`] The name of the model to remove

See also:

`restore`

`store`

`pop`

restore(*name*)

Returns the restored model

Parameters

name

[`str`] The name of the model to restore

See also:

`remove`

`store`

`pop`

store(*model*, *name=None*)

If the given model was created from this signal, stores it

Parameters

model

[`BaseModel` (or subclass)] The model to store in the signal

name

[`str` or `None`] The name for the model to be stored with

See also:

remove
restore
pop

CommonSignal1D

class hyperspy._signals.common_signal1d.CommonSignal1D

Bases: `object`

Common functions for 1-dimensional signals.

to_signal2D(*optimize=True*)

Returns the one dimensional signal as a two dimensional signal.

By default ensures the data is stored optimally, hence often making a copy of the data. See *transpose* for a more general method with more options.

optimize

[bool] If `True`, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if `True`, the chunks are optimised for the new axes configuration.

Raises

DataDimensionError
 When `data.ndim < 2`

See also:

hyperspy.api.signals.BaseSignal.as_signal2D
hyperspy.api.signals.BaseSignal.transpose
hyperspy.api.transpose

CommonSignal2D

class hyperspy._signals.common_signal2d.CommonSignal2D

Bases: `object`

Common functions for 2-dimensional signals.

to_signal1D(*optimize=True*)

Returns the image as a spectrum.

optimize

[bool] If `True`, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the *Transposing (changing signal spaces)* section of the HyperSpy user guide for more information. When operating on lazy signals, if `True`, the chunks are optimised for the new axes configuration.

See also:

hyperspy.api.signals.BaseSignal.as_signal1D
hyperspy.api.signals.BaseSignal.transpose
hyperspy.api.transpose

LazyComplexSignal

class hyperspy._lazy_signals.LazyComplexSignal(*args, **kwargs)

Bases: *ComplexSignal*, *LazySignal*

Lazy general signal class for complex data. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.ComplexSignal(data).as_lazy()
```

Create a signal instance.

Parameters

data

[*numpy.ndarray*] The signal data. It can be an array of any dimensions.

axes

[[*dict/axes*], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[*dict*, optional] A dictionary whose items are stored as attributes.

metadata

[*dict*, optional] A dictionary containing a set of parameters that will to stores in the metadata attribute. Some parameters might be mandatory in some cases.

original_metadata

[*dict*, optional] A dictionary containing a set of parameters that will to stores in the *original_metadata* attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[*bool* or *None*, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the *attributes* dictionary. If *None*, it does nothing. Default is *None*.

LazyComplexSignal1D

class hyperspy._lazy_signals.LazyComplexSignal1D(*args, **kwargs)

Bases: *ComplexSignal1D*, *LazyComplexSignal*

Lazy signal class for complex 1-dimensional data. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.ComplexSignal1D(data).as_lazy()
```

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the `ragged` value in the `attributes` dictionary. If `None`, it does nothing. Default is `None`.

LazyComplexSignal2D

```
class hyperspy._lazy_signals.LazyComplexSignal2D(*args, **kw)
```

Bases: *ComplexSignal2D*, *LazyComplexSignal*

Lazy Signal class for complex 2-dimensional data. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.ComplexSignal2D(data).as_lazy()
```

Create a signal instance.

Parameters**data**

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[*bool* or *None*, optional] Define whether the signal is ragged or not. Overwrite the *ragged* value in the *attributes* dictionary. If *None*, it does nothing. Default is *None*.

LazySignal

```
class hyperspy._signals.lazy.LazySignal(*args, **kwargs)
```

Bases: *BaseSignal*

Lazy general signal class. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.BaseSignal(data).as_lazy()
```

Create a signal instance.

Parameters**data**

[*numpy.ndarray*] The signal data. It can be an array of any dimensions.

axes

[[*dict/axes*], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[*dict*, optional] A dictionary whose items are stored as attributes.

metadata

[*dict*, optional] A dictionary containing a set of parameters that will be stored in the *metadata* attribute. Some parameters might be mandatory in some cases.

original_metadata

[*dict*, optional] A dictionary containing a set of parameters that will be stored in the *original_metadata* attribute. It typically contains all the parameters that have been imported from the original data file.

ragged

[*bool* or *None*, optional] Define whether the signal is ragged or not. Overwrite the *ragged* value in the *attributes* dictionary. If *None*, it does nothing. Default is *None*.

```
change_dtype(dtype, rechunk=False)
```

Change the data type of a Signal.

Parameters**dtype**

[*str* or *numpy.dtype*] Typecode string or data-type to which the Signal's data array is cast. In addition to all the standard *numpy Data type objects (dtype)*, HyperSpy supports four extra dtypes for RGB images: 'rgb8', 'rgba8', 'rgb16', and 'rgba16'. Changing from and to any *rgb(a) dtype* is more constrained than most other *dtype* conversions. To change to an *rgb(a) dtype*, the *signal_dimension* must be 1, and its size should be 3 (for *rgb*) or 4 (for *rgba dtypes*). The original *dtype* should be *uint8* or *uint16* if converting to *rgb(a)8* or *rgb(a)16*, and the *navigation_dimension* should be at least 2. After conversion, the *signal_dimension* becomes 2. The *dtype* of images with original *dtype* *rgb(a)8* or *rgb(a)16* can only be changed to *uint8* or *uint16*, and the *signal_dimension* becomes 1.

rechunk

[bool] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

Examples

```
>>> s = hs.signals.Signal1D([1, 2, 3, 4, 5])
>>> s.data
array([1, 2, 3, 4, 5])
>>> s.change_dtype('float')
>>> s.data
array([1., 2., 3., 4., 5.])
```

close_file()

Closes the associated data file if any.

Currently it only supports closing the file associated with a dask array created from an h5py DataSet (default HyperSpy hdf5 reader).

compute(*close_file=False, show_progressbar=None, **kwargs*)

Attempt to store the full signal in memory.

Parameters**close_file**

[bool, default `False`] If `True`, attempt to close the file associated with the dask array data if any. Note that closing the file will make all other associated lazy signals inoperative.

show_progressbar

[`None` or bool] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

****kwargs**

[dict] Any other keyword arguments for `dask.array.Array.compute()`. For example `scheduler` or `num_workers`.

Returns

`None`

Notes

For alternative ways to set the compute settings see <https://docs.dask.org/en/stable/scheduling.html#configuration>

Examples

```
>>> import dask.array as da
>>> data = da.zeros((100, 100, 100), chunks=(10, 20, 20))
>>> s = hs.signals.Signal2D(data).as_lazy()
```

With default parameters

```
>>> s1 = s.deepcopy()
>>> s1.compute()
```

Using 2 workers, which can reduce the memory usage (depending on the data and your computer hardware). Note that `num_workers` only work for the ‘threads’ and ‘processes’ scheduler.

```
>>> s2 = s.deepcopy()
>>> s2.compute(num_workers=2)
```

Using a single threaded scheduler, which is useful for debugging

```
>>> s3 = s.deepcopy()
>>> s3.compute(scheduler='single-threaded')
```

compute_navigator(*index=None, chunks_number=None, show_progressbar=None*)

Compute the navigator by taking the sum over a single chunk contained the specified coordinate. Taking the sum over a single chunk is a computationally efficient approach to compute the navigator. The data can be rechunk by specifying the `chunks_number` argument.

Parameters

index

[(`int`, `float`, `None`) or `iterable`, optional] Specified where to take the sum, follows HyperSpy indexing syntax for integer and float. If `None`, the index is the centre of the `signal_space`

chunks_number

[(`int`, `None`) or `iterable`, optional] Define the number of chunks in the signal space used for rechunk the when calculating of the navigator. Useful to define the range over which the sum is calculated. If `None`, the existing chunking will be considered when picking the chunk used in the navigator calculation.

show_progressbar

[`None` or `bool`] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

Returns

`None`.

Notes

The number of chunks will affect where the sum is taken. If the sum needs to be taken in the centre of the signal space (for example, in the case of diffraction pattern), the number of chunk needs to be an odd number, so that the middle is centered.

decomposition(*normalize_poissonian_noise=False, algorithm='SVD', output_dimension=None, signal_mask=None, navigation_mask=None, get=None, num_chunks=None, reproject=True, print_info=True, **kwargs*)

Perform Incremental (Batch) decomposition on the data.

The results are stored in the `learning_results` attribute.

Read more in the *User Guide*.

Parameters

normalize_poissonian_noise

[`bool`, default `False`] If True, scale the signal to normalize Poissonian noise using the approach described in [KeenanKotula2004].

algorithm

[{'SVD', 'PCA', 'ORPCA', 'ORNMF'}, default 'SVD'] The decomposition algorithm to use.

output_dimension

[`int` or `None`, default `None`] Number of components to keep/calculate. If `None`, keep all (only valid for 'SVD' algorithm)

get

[`dask scheduler` or `None`] The dask scheduler to use for computations. If `None`, `dask.threaded.get`` will be used if possible, otherwise ```dask.get` will be used, for example in pyodide interpreter.

num_chunks

[`int` or `None`, default `None`] the number of dask chunks to pass to the decomposition model. More chunks require more memory, but should run faster. Will be increased to contain at least `output_dimension` signals.

navigation_mask

[`class:~.api.signals.BaseSignal, numpy.ndarray` or `dask.array.Array`] The navigation locations marked as True are not used in the decomposition. Not implemented for the 'SVD' algorithm.

signal_mask

[`class:~.api.signals.BaseSignal, numpy.ndarray` or `dask.array.Array`] The signal locations marked as True are not used in the decomposition. Not implemented for the 'SVD' algorithm.

reproject

[`bool`, default `True`] Reproject data on the learnt components (factors) after learning.

print_info

[`bool`, default `True`] If True, print information about the decomposition being performed. In the case of `sklearn.decomposition` objects, this includes the values of all arguments of the chosen `sklearn` algorithm.

****kwargs**

passed to the `partial_fit/fit` functions.

See also:

`dask.array.linalg.svd`, `sklearn.decomposition.IncrementalPCA`
`hyperspy.learn.rpca.ORPCA`, `hyperspy.learn.ornmf.ORNMF`

References

[KeenanKotula2004]

diff(*axis*, *order=1*, *out=None*, *rechunk=False*)

Returns a signal with the n-th order discrete difference along given axis. *i.e.* it calculates the difference between consecutive values in the given axis: $out[n] = a[n+1] - a[n]$. See `numpy.diff()` for more details.

Parameters

axis

[`int`, `str`, or `DataAxis`] The axis can be passed directly, or specified using the index of the axis in the Signal's `axes_manager` or the axis name.

order

[`int`] The order of the discrete difference.

out

[`BaseSignal` (or subclass) or `None`] If `None`, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[`bool`] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

Returns

`BaseSignal` or `None`

Note that the size of the data on the given axis decreases by the given order. *i.e.* if axis is "x" and order is 2, the x dimension is N, der's x dimension is N - 2.

See also:

`hyperspy.api.signals.BaseSignal.derivative`
`hyperspy.api.signals.BaseSignal.integrate1D`
`hyperspy.api.signals.BaseSignal.integrate_simpson`

Notes

If you intend to calculate the numerical derivative, please use the proper `derivative()` function instead. To avoid erroneous misuse of the `diff` function as derivative, it raises an error when working with a non-uniform axis.

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.diff(0)
<BaseSignal, title: , dimensions: (|1023, 64, 64)>
```

`get_chunk_size`(*axes=None*)

Returns the chunk size as tuple for a set of given axes. The order of the returned tuple follows the order of the dask array.

Parameters

axes

[*int*, *str*, *DataAxis* or *tuple*] Either one on its own, or many axes in a tuple can be passed. In both cases the axes can be passed directly, or specified using the index in *axes_manager* or the name of the axis. Any duplicates are removed. If *None*, the operation is performed over all navigation axes (default).

Examples

```
>>> import dask.array as da
>>> data = da.random.random((10, 200, 300))
>>> data.chunksize
(10, 200, 300)
>>> s = hs.signals.Signal1D(data).as_lazy()
>>> s.get_chunk_size() # All navigation axes
((10,), (200,))
>>> s.get_chunk_size(0) # The first navigation axis
((200,),)
```

`get_histogram`(*bins='fd'*, *out=None*, *rechunk=False*, ***kwargs*)

Return a histogram of the signal data.

More sophisticated algorithms for determining the bins can be used by passing a string as the *bins* argument. Other than the 'blocks' and 'knuth' methods, the available algorithms are the same as `numpy.histogram()`.

Note: The lazy version of the algorithm only supports "scott" and "fd" as a string argument for *bins*.

Parameters

bins

[*int* or *sequence of float* or *str*, default "fd"] If *bins* is an *int*, it defines the number of equal-width bins in the given range. If *bins* is a *sequence*, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If *bins* is a *string* from the list below, will use the method chosen to calculate the optimal bin width and consequently the number of bins (see Notes for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the 'auto' option is suggested. Weighted data is not supported for automated bin size selection.

‘auto’

Maximum of the ‘sturges’ and ‘fd’ estimators. Provides good all around performance.

‘fd’ (Freedman Diaconis Estimator)

Robust (resilient to outliers) estimator that takes into account data variability and data size.

‘doane’

An improved version of Sturges’ estimator that works better with non-normal datasets.

‘scott’

Less robust estimator that that takes into account data variability and data size.

‘stone’

Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott’s rule.

‘rice’

Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

‘sturges’

R’s default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.

‘sqrt’

Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.

‘knuth’

Knuth’s rule is a fixed-width, Bayesian approach to determining the optimal bin width of a histogram.

‘blocks’

Determination of optimal adaptive-width histogram bins using the Bayesian Blocks algorithm.

range_bins

[[tuple](#) or [None](#), optional] the minimum and maximum range for the histogram. If *range_bins* is [None](#), (`x.min()`, `x.max()`) will be used.

max_num_bins

[[int](#), default 250] When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a [MemoryError](#) being raised by [numpy.histogram\(\)](#).

out

[[BaseSignal](#) (or subclass) or [None](#)] If [None](#), a new [Signal](#) is created with the result of the operation and returned (default). If a [Signal](#) is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[[bool](#)] Only has effect when operating on lazy signal. Default [False](#), which means the chunking structure will be retained. If [True](#), the data may be automatically rechunked before performing this operation.

****kwargs**

other keyword arguments (weight and density) are described in [numpy.histogram\(\)](#).

Returns**hist_spec**

[*Signal1D*] A 1D spectrum instance containing the histogram.

See also:

hyperspy.api.signals.BaseSignal.print_summary_statistics
numpy.histogram, *dask.array.histogram*

Examples

```
>>> s = hs.signals.Signal1D(np.random.normal(size=(10, 100)))
>>> # Plot the data histogram
>>> s.get_histogram().plot()
>>> # Plot the histogram of the signal at the current coordinates
>>> s.get_current_signal().get_histogram().plot()
```

integrate_simpson(*axis*, *out=None*, *rechunk=False*)

Calculate the integral of a Signal along an axis using Simpson's rule.

Parameters**axis**

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns**s**

[*BaseSignal* (or subclass)] A new Signal containing the integral of the provided Signal along the specified axis.

See also:

hyperspy.api.signals.BaseSignal.derivative
hyperspy.api.signals.BaseSignal.integrate1D

Examples

```
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.integrate_simpson(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

plot(navigator='auto', **kwargs)

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

navigator

[str, None, or *BaseSignal* (or subclass).]

Allowed string values are ``'auto'``, ``'slider'``, and ``'spectrum'``.

- If 'auto':
 - If navigation_dimension > 0, a navigator is provided to explore the data.
 - If navigation_dimension is 1 and the signal is an image the navigator is a sum spectrum obtained by integrating over the signal axes (the image).
 - If navigation_dimension is 1 and the signal is a spectrum the navigator is an image obtained by stacking all the spectra in the dataset horizontally.
 - If navigation_dimension is > 1, the navigator is a sum image obtained by integrating the data over the signal axes.
 - Additionally, if navigation_dimension > 2, a window with one slider per axis is raised to navigate the data.
 - For example, if the dataset consists of 3 navigation axes “X”, “Y”, “Z” and one signal axis, “E”, the default navigator will be an image obtained by integrating the data over “E” at the current “Z” index and a window with sliders for the “X”, “Y”, and “Z” axes will be raised. Notice that changing the “Z”-axis index changes the navigator in this case.
 - For lazy signals, the navigator will be calculated using the `compute_navigator()` method.
- If 'slider':
 - If navigation_dimension > 0 a window with one slider per axis is raised to navigate the data.
- If 'spectrum':
 - If navigation_dimension > 0 the navigator is always a spectrum obtained by integrating the data over all other axes.
 - Not supported for lazy signals, the 'auto' option will be used instead.
- If None, no navigator will be provided.

Alternatively a *BaseSignal* (or subclass) instance can be provided. The navigation or signal shape must match the navigation shape of the signal to plot or the

navigation_shape + signal_shape must be equal to the navigator_shape of the current object (for a dynamic navigator). If the signal dtype is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

axes_manager

[None or *AxesManager*] If None, the signal's axes_manager attribute is used.

plot_markers

[bool, default True] Plot markers added using *s.add_marker(marker, permanent=True)*. Note, a large number of markers might lead to very slow plotting.

navigator_kwds

[dict] Only for image navigator, additional keyword arguments for *matplotlib.pyplot.imshow()*.

norm

[str, default 'auto'] The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. If 'auto', intensity is plotted on a linear scale except when *power_spectrum=True* (only for complex signals).

autoscale

[str] The string must contain any combination of the 'x' and 'v' characters. If 'x' or 'v' (for values) are in the string, the corresponding horizontal or vertical axis limits are set to their maxima and the axis limits will reset when the data or the navigation indices are changed. Default is 'v'.

**kwargs

[dict] Only when plotting an image: additional (optional) keyword arguments for *matplotlib.pyplot.imshow()*.

rebin(*new_shape=None, scale=None, crop=False, dtype=None, out=None, rechunk=False*)

Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify **either** *new_shape* or *scale*. Scale of 1 means no binning and scale less than one results in up-sampling.

Parameters

new_shape

[list (of float or int) or None] For each dimension specify the new_shape. This will internally be converted into a scale parameter.

scale

[list (of float or int) or None] For each dimension, specify the new:old pixel ratio, e.g. a ratio of 1 is no binning and a ratio of 2 means that each pixel in the new spectrum is twice the size of the pixels in the old spectrum. The length of the list should match the dimension of the Signal's underlying data array. *Note : Only one of ``scale`` or ``new_shape`` should be specified, otherwise the function will not run*

crop

[bool] Whether or not to crop the resulting rebinned data (default is True). When binning by a non-integer number of pixels it is likely that the final row in each dimension will contain fewer than the full quota to fill one pixel. For example, a 5*5 array binned by 2.1 will produce two rows containing 2.1 pixels and one row containing only 0.8 pixels. Selection of *crop=True* or *crop=False* determines whether or not this "black" line is cropped from the final binned array or not. *Please note that if ``crop=False`` is used, the final row in each dimension may appear black if a fractional number of pixels are left over. It can be removed but has been left to preserve total counts before and after binning.*

dtype

[{None, numpy.dtype, "same"}] Specify the dtype of the output. If None, the dtype

will be determined by the behaviour of `numpy.sum()`, if "same", the dtype will be kept the same. Default is None.

out

[*BaseSignal* (or subclass) or `None`] If `None`, a new `Signal` is created with the result of the operation and returned (default). If a `Signal` is passed, it is used to receive the output of the operation, and nothing is returned.

Returns*BaseSignal*

The resulting cropped signal.

Raises`NotImplementedError`

If trying to rebin over a non-uniform axis.

Examples

```
>>> spectrum = hs.signals.Signal1D(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<Signal1D, title: , dimensions: (4, 4|10)>
>>> print('Sum =', sum(sum(sum(spectrum.data))))
Sum = 164.0
```

```
>>> scale = [2, 2, 5]
>>> test = spectrum.rebin(scale)
>>> print(test)
<Signal1D, title: , dimensions: (2, 2|5)>
>>> print('Sum =', sum(sum(sum(test.data))))
Sum = 164.0
```

```
>>> s = hs.signals.Signal1D(np.ones((2, 5, 10), dtype=np.uint8))
>>> print(s)
<Signal1D, title: , dimensions: (5, 2|10)>
>>> print(s.data.dtype)
uint8
```

Use `dtype=np.uint16` to specify a dtype

```
>>> s2 = s.rebin(scale=(5, 2, 1), dtype=np.uint16)
>>> print(s2.data.dtype)
uint16
```

Use `dtype="same"` to keep the same dtype

```
>>> s3 = s.rebin(scale=(5, 2, 1), dtype="same")
>>> print(s3.data.dtype)
uint8
```

By default `dtype=None`, the dtype is determined by the behaviour of `numpy.sum`, in this case, unsigned integer of the same precision as the platform integer

```
>>> s4 = s.rebin(scale=(5, 2, 1))
>>> print(s4.data.dtype)
uint32
```

rechunk(*nav_chunks='auto', sig_chunks=-1, inplace=True, **kwargs*)

Rechunks the data using the same rechunking formula from Dask except that the navigation and signal chunks are defined separately. Note, for most functions *sig_chunks* should remain `None` so that it spans the entire signal axes.

Parameters

nav_chunks

[`{tuple, int, "auto", None}`] The navigation block dimensions to create. -1 indicates the full size of the corresponding dimension. Default is "auto" which automatically determines chunk sizes.

sig_chunks

[`{tuple, int, "auto", None}`] The signal block dimensions to create. -1 indicates the full size of the corresponding dimension. Default is -1 which automatically spans the full signal dimension

****kwargs**

[`dict`] Any other keyword arguments for `dask.array.rechunk()`.

valuemax(*axis, out=None, rechunk=False*)

Returns a signal with the value of coordinates of the maximum along an axis.

Parameters

axis

[`int, str, or DataAxis`] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[`BaseSignal` (or subclass) or `None`] If `None`, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[`bool`] Only has effect when operating on lazy signal. Default `False`, which means the chunking structure will be retained. If `True`, the data may be automatically rechunked before performing this operation.

Returns

s

[`BaseSignal` (or subclass)] A new Signal containing the calibrated coordinate values of the maximum along the specified axis.

See also:

`hyperspy.api.signals.BaseSignal.max`, `hyperspy.api.signals.BaseSignal.min`
`hyperspy.api.signals.BaseSignal.sum`, `hyperspy.api.signals.BaseSignal.mean`
`hyperspy.api.signals.BaseSignal.std`, `hyperspy.api.signals.BaseSignal.var`
`hyperspy.api.signals.BaseSignal.indexmax`,
`hyperspy.api.signals.BaseSignal.indexmin`
`hyperspy.api.signals.BaseSignal.valuemin`

Examples

```
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s
<BaseSignal, title: , dimensions: (|1024, 64, 64)>
>>> s.valuemax(0)
<Signal2D, title: , dimensions: (|64, 64)>
```

valuemin(*axis*, *out=None*, *rechunk=False*)

Returns a signal with the value of coordinates of the minimum along an axis.

Parameters

axis

[*int*, *str*, or *DataAxis*] The axis can be passed directly, or specified using the index of the axis in the Signal's *axes_manager* or the axis name.

out

[*BaseSignal* (or subclass) or *None*] If *None*, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

rechunk

[*bool*] Only has effect when operating on lazy signal. Default *False*, which means the chunking structure will be retained. If *True*, the data may be automatically rechunked before performing this operation.

Returns

BaseSignal or subclass

A new Signal containing the calibrated coordinate values of the minimum along the specified axis.

See also:

hyperspy.api.signals.BaseSignal.max, *hyperspy.api.signals.BaseSignal.min*
hyperspy.api.signals.BaseSignal.sum, *hyperspy.api.signals.BaseSignal.mean*
hyperspy.api.signals.BaseSignal.std, *hyperspy.api.signals.BaseSignal.var*
hyperspy.api.signals.BaseSignal.indexmax,
hyperspy.api.signals.BaseSignal.indexmin
hyperspy.api.signals.BaseSignal.valuemax

LazySignal1D

class hyperspy._lazy_signals.LazySignal1D(*args, **kwargs)

Bases: *LazySignal*, *Signal1D*

Lazy general 1D signal class. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.Signal1D(data).as_lazy()
```

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[`bool` or `None`, optional] Define whether the signal is ragged or not. Overwrite the `ragged` value in the `attributes` dictionary. If `None`, it does nothing. Default is `None`.

LazySignal2D

```
class hyperspy._lazy_signals.LazySignal2D(*args, **kwargs)
```

Bases: *LazySignal*, *Signal2D*

Lazy general 2D signal class. The computation is delayed until explicitly requested.

This class is not expected to be instantiated directly, instead use:

```
>>> data = da.ones((10, 10))
>>> s = hs.signals.Signal2D(data).as_lazy()
```

Create a signal instance.

Parameters

data

[`numpy.ndarray`] The signal data. It can be an array of any dimensions.

axes

[[`dict/axes`], optional] List of either dictionaries or axes objects to define the axes (see the documentation of the *AxesManager* class for more details).

attributes

[`dict`, optional] A dictionary whose items are stored as attributes.

metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `metadata` attribute. Some parameters might be mandatory in some cases.

original_metadata

[`dict`, optional] A dictionary containing a set of parameters that will to stores in the `original_metadata` attribute. It typically contains all the parameters that has been imported from the original data file.

ragged

[*bool* or *None*, optional] Define whether the signal is ragged or not. Overwrite the ragged value in the `attributes` dictionary. If *None*, it does nothing. Default is *None*.

23.9.6 ROI

<i>BaseROI()</i>	Base class for all ROIs.
<i>BaseInteractiveROI()</i>	Base class for interactive ROIs, i.e. ROIs with widget interaction.

Region of interests (ROIs).

ROIs operate on *BaseSignal* instances and include widgets for interactive operation.

The following 1D ROIs are available:

Point1DROI

Single element ROI of a 1D signal.

SpanROI

Interval ROI of a 1D signal.

The following 2D ROIs are available:

Point2DROI

Single element ROI of a 2D signal.

RectangularROI

Rectangular ROI of a 2D signal.

CircleROI

(Hollow) circular ROI of a 2D signal

Line2DROI

Line profile of a 2D signal with customisable width.

class hyperspy.roi.BaseInteractiveROI

Bases: *BaseROI*

Base class for interactive ROIs, i.e. ROIs with widget interaction. The base class defines a lot of the common code for interacting with widgets, but inheritors need to implement the following functions:

`_get_widget_type()` `_apply_roi2widget(widget)` `_set_from_widget(widget)`

Sets up `events.changed` event, and inits `HasTraits`.

add_widget(*signal*, *axes=None*, *widget=None*, *color='green'*, *snap=True*, ***kwargs*)

Add a widget to visually represent the ROI, and connect it so any changes in either are reflected in the other. Note that only one widget can be added per signal/axes combination.

Parameters**signal**

[*hyperspy.api.signals.BaseSignal* (or subclass)] The signal to which the widget is added. This is used to determine which plot to add the widget to, and it supplies the `axes_manager` for the widget.

axes

[*None*, *str*, *int* or *hyperspy.axes.DataAxis*, default *None*] The axes argument

specifies which axes the ROI will be applied on. The axes in the collection can be either of the following:

- Anything that can index the provided `axes_manager`.
- a tuple or list of:
 - `hyperspy.axes.DataAxis`
 - anything that can index the provided `axes_manager`
- None, it will check whether the widget can be added to the navigator, i.e. if dimensionality matches, and use it if possible, otherwise it will try the signal space. If none of the two attempts work, an error message will be raised.

widget

[hyperspy widget or `None`, default `None`] If specified, this is the widget that will be added. If `None`, the default widget will be used.

color

[matplotlib color, default 'green'] The color for the widget. Any format that matplotlib uses should be ok. This will not change the color for any widget passed with the 'widget' argument.

snap

[bool, default `True`] If `True`, the ROI will be snapped to the axes values.

****kwargs**

[dict] All keyword arguments are passed to the widget constructor.

Returns

hyperspy widget

The widget of the ROI.

interactive(*signal*, *navigation_signal*='same', *out*=None, *color*='green', *snap*=True, ***kwargs*)

Creates an interactively sliced Signal (sliced by this ROI) via `interactive()`.

Parameters

signal

[`hyperspy.api.signals.BaseSignal` (or subclass)] The source signal to slice.

navigation_signal

[`hyperspy.api.signals.BaseSignal` (or subclass), `None` or "same" (default)] The signal the ROI will be added to, for navigation purposes only. Only the source signal will be sliced. If not `None`, it will automatically create a widget on `navigation_signal`. Passing "same" is identical to passing the same signal to "signal" and "navigation_signal", but is less ambiguous, and allows "same" to be the default value.

out

[`hyperspy.api.signals.BaseSignal` (or subclass)] If not `None`, it will use 'out' as the output instead of returning a new Signal.

color

[matplotlib color, default: 'green'] The color for the widget. Any format that matplotlib uses should be ok. This will not change the color for any widget passed with the 'widget' argument.

snap

[bool, default `True`] If `True`, the ROI will be snapped to the axes values.

****kwargs**

All kwargs are passed to the `roi.__call__` method which is called interactively on any roi parameter change.

Returns

BaseSignal (or subclass)

Signal updated with the current ROI selection when the ROI is changed.

remove_widget(*signal*, *render_figure=True*)

Removing a widget from a signal consists of two tasks:

1. Disconnect the interactive operations associated with this ROI and the specified signal `signal`.
2. Removing the widget from the plot.

Parameters

signal

[*hyperspy.api.signals.BaseSignal* (or subclass)] The signal from which the interactive operations will be disconnected.

render_figure

[*bool*, default *True*] If *False*, the figure will not be rendered after removing the widget in order to save redraw events.

update()

Function responsible for updating anything that depends on the ROI. It should be called by implementors whenever the ROI changes. This implementation updates the widgets associated with it, and triggers the changed event.

class hyperspy.roi.BaseROI

Bases: *HasTraits*

Base class for all ROIs.

Provides some basic functionalities that are likely to be shared between all ROIs, and serve as a common type that can be checked for.

Sets up `events.changed` event, and inits *HasTraits*.

is_valid()

Determine if the ROI is in a valid state.

This is typically determined by all the coordinates being defined, and that the values makes sense relative to each other.

update()

Function responsible for updating anything that depends on the ROI. It should be called by implementors whenever the ROI changes. The base implementation simply triggers the changed event.

23.10 Utils

API of classes, which are not part of the *hyperspy.api* namespace but used by HyperSpy signals.

class `hyperspy.misc.utils.DictionaryTreeBrowser`(*dictionary=None, double_lines=False, lazy=True*)

Bases: `object`

A class to comfortably browse a dictionary using a CLI.

In addition to accessing the values using dictionary syntax the class enables navigating a dictionary that contains nested dictionaries as attributes of nested classes. Also it is an iterator over the (key, value) items. The `__repr__` method provides pretty tree printing. Private keys, i.e. keys that starts with an underscore, are not printed, counted when calling `len` nor iterated.

Examples

```
>>> tree = DictionaryTreeBrowser()
>>> tree.set_item("Branch.Leaf1.color", "green")
>>> tree.set_item("Branch.Leaf2.color", "brown")
>>> tree.set_item("Branch.Leaf2.caterpillar", True)
>>> tree.set_item("Branch.Leaf1.caterpillar", False)
>>> tree
├─ Branch
│   ├── Leaf1
│   │   ├── caterpillar = False
│   │   └─ color = green
│   └─ Leaf2
│       ├── caterpillar = True
│       └─ color = brown
>>> tree.Branch
├─ Leaf1
│   ├── caterpillar = False
│   └─ color = green
└─ Leaf2
    ├── caterpillar = True
    └─ color = brown
>>> for label, leaf in tree.Branch:
...     print("%s is %s" % (label, leaf.color))
Leaf1 is green
Leaf2 is brown
>>> tree.Branch.Leaf2.caterpillar
True
>>> "Leaf1" in tree.Branch
True
>>> "Leaf3" in tree.Branch
False
>>>
```

add_dictionary(*dictionary, double_lines=False*)

Add new items from dictionary.

add_node(*node_path*)

Adds all the nodes in the given path if they don't exist.

Parameters

node_path: str

The nodes must be separated by full stops (periods).

Examples

```
>>> dict_browser = DictionaryTreeBrowser({})
>>> dict_browser.add_node('First.Second')
>>> dict_browser.First.Second = 3
>>> dict_browser
└─ First
   └─ Second = 3
```

as_dictionary()

Returns its dictionary representation.

copy()

Returns a shallow copy using `copy.copy()`.

deepcopy()

Returns a deep copy using `copy.deepcopy()`.

export(filename, encoding='utf8')

Export the dictionary to a text file

Parameters

filename

[str] The name of the file without the extension that is txt by default

encoding

[str] The encoding to be used.

get_item(item_path, default=None, full_path=True, wild=False, return_path=False)

Given a path, return it's value if it exists, or default value if missing. May also perform a search whether an item key exists and then returns the value or a list of values for multiple occurrences of the key – optionally returns the full path(s) in addition to its value(s).

The nodes of the path are separated using periods.

Parameters

item_path

[str] A string describing the path with each item separated by full stops (periods)

full_path

[bool, default True] If True, the full path to the item has to be given. If False, a search for the item key is performed (can include additional nodes preceding they key separated by full stops).

wild

[bool] Only applies if `full_path=False`. If True, searches for any items where `item` matches a substring of the item key (case insensitive). Default is False.

return_path

[bool] Only applies if `full_path=False`. Default False. If True, returns an additional list of paths to the item(s) that match key.

default

[None or object, default None] The value to return if the path or item does not exist.

Examples

```
>>> dict = {'To' : {'be' : True}}
>>> dict_browser = DictionaryTreeBrowser(dict)
>>> dict_browser.get_item('To')
└─ be = True
>>> dict_browser.get_item('To.be')
True
>>> dict_browser.get_item('To.be.or', 'default_value')
'default_value'
>>> dict_browser.get_item('be', full_path=False)
True
```

has_item(*item_path*, *default=None*, *full_path=True*, *wild=False*, *return_path=False*)

Given a path, return True if it exists. May also perform a search whether an item exists and optionally returns the full path instead of boolean value.

The nodes of the path are separated using periods.

Parameters**item_path**

[str] A string describing the path with each item separated by full stops (periods).

full_path

[bool, default True] If True, the full path to the item has to be given. If False, a search for the item key is performed (can include additional nodes preceding they key separated by full stops).

wild

[bool, default True] Only applies if `full_path=False`. If True, searches for any items where `item` matches a substring of the item key (case insensitive). Default is False.

return_path

[bool, default False] Only applies if `full_path=False`. If False, a boolean value is returned. If True, the full path to the item is returned, a list of paths for multiple matches, or default value if it does not exist.

default

The value to return for path if the item does not exist (default is None).

Examples

```
>>> dict = {'To' : {'be' : True}}
>>> dict_browser = DictionaryTreeBrowser(dict)
>>> dict_browser.has_item('To')
True
>>> dict_browser.has_item('To.be')
True
>>> dict_browser.has_item('To.be.or')
```

(continues on next page)

(continued from previous page)

```
False
>>> dict_browser.has_item('be', full_path=False)
True
>>> dict_browser.has_item('be', full_path=False, return_path=True)
'To.be'
```

keys()

Returns a list of non-private keys.

process_lazy_attributes()

Run the DictionaryTreeBrowser machinery for the lazy attributes.

set_item(item_path, value)

Given the path and value, create the missing nodes in the path and assign the given value.

Parameters**item_path**

[str] A string describing the path with each item separated by a full stop (periods)

value

[object] The value to assign to the given path.

Examples

```
>>> dict_browser = DictionaryTreeBrowser({})
>>> dict_browser.set_item('First.Second.Third', 3)
>>> dict_browser
├─ First
│   └─ Second
│       └─ Third = 3
```

`hyperspy.misc.export_dictionary.export_to_dictionary(target, whitelist, dic, fullcopy=True)`

Exports attributes of target from `whitelist.keys()` to dictionary `dic`. All values are references only by default.

Parameters**target**

[object] must contain the (nested) attributes of the `whitelist.keys()`

whitelist

[dict] A dictionary, keys of which are used as attributes for exporting. Key 'self' is only available with tag 'id', when the id of the target is saved. The values are either None, or a tuple, where:

- the first item a string, which contains flags, separated by commas.
- the second item is None if no 'init' flag is given, otherwise the object required for the initialization.

The flag conventions are as follows:

- 'init': object used for initialization of the target. The object is saved in the tuple in `whitelist`
- 'fn': the targeted attribute is a function, and may be pickled. A tuple of (thing, value) will be exported to the dictionary, where thing is None if function is passed as-is, and

True if cloudpickle package is used to pickle the function, with the value as the result of the pickle.

- 'id': the id of the targeted attribute is exported (e.g. `id(target.name)`)
- 'sig': The targeted attribute is a signal, and will be converted to a dictionary if `full-copy=True`

dict

A dictionary where the object will be exported

bool

Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

23.10.1 peakfinders2D

`hyperspy.utils.peakfinders2D.clean_peaks(peaks)`

Sort array of peaks and deal with no peaks being found.

Parameters

peaks

[`numpy.ndarray`] Array of found peaks.

Returns

peaks

[`numpy.ndarray`] Sorted array, first by `peaks[:,1]` (y-coordinate) then by `peaks[:,0]` (x-coordinate), of found peaks.

NO_PEAKS

[`str`] Flag indicating no peaks found.

`hyperspy.utils.peakfinders2D.find_local_max(z, **kwargs)`

Method to locate positive peaks in an image by local maximum searching.

This function wraps `skimage.feature.peak_local_max()` function and sorts the results for consistency with other peak finding methods.

Parameters

z

[`numpy.ndarray`] Array of image intensities.

****kwargs**

[`dict`] Keyword arguments to be passed to the `skimage.feature.peak_local_max()` function.

Returns

`numpy.ndarray`

Peak pixel coordinates with shape (n_peaks, 2).

`hyperspy.utils.peakfinders2D.find_peaks_dog(z, min_sigma=1.0, max_sigma=50.0, sigma_ratio=1.6, threshold=0.2, overlap=0.5, exclude_border=False)`

Method to locate peaks via the Difference of Gaussian Matrices method.

This function wraps `skimage.feature.blob_dog()` function and sorts the results for consistency with other peak finding methods.

Parameters

z
[`numpy.ndarray`] 2-d array of intensities

min_sigma, max_sigma, sigma_ratio, threshold, overlap, exclude_border
Additional parameters to be passed to the `skimage.feature.blob_dog()` function

Returns

peaks
[`numpy.ndarray`] Peak pixel coordinates with shape (n_peaks, 2).

Notes

While highly effective at finding even very faint peaks, this method is sensitive to fluctuations in intensity near the edges of the image.

`hyperspy.utils.peakfinders2D.find_peaks_log(z, min_sigma=1.0, max_sigma=50.0, num_sigma=10, threshold=0.2, overlap=0.5, log_scale=False, exclude_border=False)`

Method to locate peaks via the Laplacian of Gaussian Matrices method.

This function wraps `skimage.feature.blob_log()` function and sorts the results for consistency with other peak finding methods.

Parameters

z
[`numpy.ndarray`] Array of image intensities.

min_sigma, max_sigma, num_sigma, threshold, overlap, log_scale, exclude_border
Additional parameters to be passed to the `skimage.feature.blob_log()` function.

Returns

peaks
[`numpy.ndarray`] Peak pixel coordinates with shape (n_peaks, 2).

`hyperspy.utils.peakfinders2D.find_peaks_max(z, alpha=3.0, distance=10)`

Method to locate positive peaks in an image by local maximum searching.

Parameters

alpha
[`float`] Only maxima above $alpha * sigma$ are found, where $sigma$ is the standard deviation of the image.

distance
[`int`] When a peak is found, all pixels in a square region of side $2 * distance$ are set to zero so that no further peaks can be found in that region.

Returns

peaks
[`numpy.ndarray`] Peak pixel coordinates with shape (n_peaks, 2).

`hyperspy.utils.peakfinders2D.find_peaks_minmax(z, distance=5.0, threshold=10.0)`

Method to locate the positive peaks in an image by comparing maximum and minimum filtered images.

Parameters

z
[`numpy.ndarray`] Matrix of image intensities.

distance

[float] Expected distance between peaks.

threshold

[float] Minimum difference between maximum and minimum filtered images.

Returns**peaks**

[numpy.ndarray] Peak pixel coordinates with shape (n_peaks, 2).

`hyperspy.utils.peakfinders2D.find_peaks_stat(z, alpha=1.0, window_radius=10, convergence_ratio=0.05)`

Method to locate positive peaks in an image based on statistical refinement and difference with respect to mean intensity.

Parameters**z**

[numpy.ndarray] Array of image intensities.

alpha

[float] Only maxima above $\alpha * \sigma$ are found, where σ is the local, rolling standard deviation of the image.

window_radius

[int] The pixel radius of the circular window for the calculation of the rolling mean and standard deviation.

convergence_ratio

[float] The algorithm will stop finding peaks when the proportion of new peaks being found is less than *convergence_ratio*.

Returns**peaks**

[numpy.ndarray] Peak pixel coordinates with with shape (n_peaks, 2).

Notes

Implemented as described in the PhD thesis of Thomas White, University of Cambridge, 2009, with minor modifications to resolve ambiguities.

The algorithm is as follows:

1. Adjust the contrast and intensity bias of the image so that all pixels have values between 0 and 1.
2. For each pixel, determine the mean and standard deviation of all pixels inside a circle of radius 10 pixels centered on that pixel.
3. If the value of the pixel is greater than the mean of the pixels in the circle by more than one standard deviation, set that pixel to have an intensity of 1. Otherwise, set the intensity to 0.
4. Smooth the image by convolving it twice with a flat 3x3 kernel.
5. Let $k = (1/2 - \mu)/\sigma$ where μ and σ are the mean and standard deviations of all the pixel intensities in the image.
6. For each pixel in the image, if the value of the pixel is greater than $\mu + k*\sigma$ set that pixel to have an intensity of 1. Otherwise, set the intensity to 0.
7. Detect peaks in the image by locating the centers of gravity of regions of adjacent pixels with a value of 1.

- Repeat #4-7 until the number of peaks found in the previous step converges to within the user defined `convergence_ratio`.

`hyperspy.utils.peakfinders2D.find_peaks_xc(z, template, distance=5, threshold=0.5, **kwargs)`

Find peaks in the cross correlation between the image and a template by using the `find_peaks_minmax()` function to find the peaks on the cross correlation result obtained using the `skimage.feature.match_template()` function.

Parameters

z

[`numpy.ndarray`] Array of image intensities.

template

[`numpy.ndarray`] Array containing a single bright disc, similar to those to detect.

distance

[`float`] Expected distance between peaks.

threshold

[`float`] Minimum difference between maximum and minimum filtered images.

****kwargs**

[`dict`] Keyword arguments to be passed to the `skimage.feature.match_template()` function.

Returns

peaks

[`numpy.ndarray`] Array of peak coordinates with shape (n_peaks, 2).

`hyperspy.utils.peakfinders2D.find_peaks_zaefferer(z, grad_threshold=0.1, window_size=40, distance_cutoff=50.0)`

Method to locate positive peaks in an image based on gradient thresholding and subsequent refinement within masked regions.

Parameters

z

[`numpy.ndarray`] Matrix of image intensities.

grad_threshold

[`float`] The minimum gradient required to begin a peak search.

window_size

[`int`] The size of the square window within which a peak search is conducted. If odd, will round down to even. The size must be larger than 2.

distance_cutoff

[`float`] The maximum distance a peak may be from the initial high-gradient point.

Returns

peaks

[`numpy.ndarray`] Peak pixel coordinates with shape (n_peaks, 2).

Notes

Implemented as described in Zaefferer “New developments of computer-aided crystallographic analysis in transmission electron microscopy” J. Ap. Cryst. This version by Ben Martineau (2016)

GETTING HELP

There are several places to obtain help with HyperSpy:

- The [HyperSpy Gitter](#) chat is a good place to go for both troubleshooting and general questions.
- Issue with installation? There are some troubleshooting tips built into the [installation page](#),
- If you want to request new features or if you're confident that you have found a bug, please create a new issue on the [HyperSpy GitHub issues](#) page. When reporting bugs, please try to replicate the bug with the HyperSpy sample data, and make every effort to simplify your example script to only the elements necessary to replicate the bug.

CHANGELOG

Changelog entries for the development version are available at <https://hyperspy.readthedocs.io/en/latest/changes.html>

25.1 2.0.2.dev70+g18caceb [UNRELEASED] (2024-04-11)

25.1.1 Enhancements

- Add an dynamic navigator which updates when the number of navigation dimensions is greater than 3 (#3199)
- Add an example to the gallery to show how to extract a line profile from an image using a Line2DROI (#3227)

25.1.2 Bug Fixes

- Fix ROI slicing of non-uniform axis (#3328)
- Adds the ability to save and load *BaseDataAxis* objects to a hyperspy file. (#3342)

25.1.3 Maintenance

- Ruff update:
 - set the `RELEASE_next_patch` branch as target for the `pre-commit.ci` update to keep all branches in synchronisation.
 - update ruff to version 0.3.3 and run `ruff check/format` on source code. (#3335)
- Replace deprecated `np.string_` by `np.bytes_`. (#3338)
- Enable ruff isort and all pyflakes/Pycodestyle rules, except E501 to avoid conflict with black formatting. (#3348)
- Convert projet readme to markdown, fixes badges on github (#3351)

25.2 2.0.1 (2024-02-26)

25.2.1 Bug Fixes

- Fix bug with side by side plotting of signal containing navigation dimension only. (#3304)
- Fix getting release on some linux system wide install, e.g. Debian or Google colab (#3318)
- Fix incorrect position of Texts marker when using mathtext. (#3319)

25.2.2 Maintenance

- Update version switcher. (#3291)
- Fix readme badges and fix broken web links. (#3298)
- Use ruff to lint code. (#3299)
- Use ruff to format code. (#3300)
- Run test suite on osx arm64 on GitHub CI and speed running test suite using all available CPUs (3 or 4) instead of only 2. (#3305)
- Fix API changes in `scipy.signal.windows.tukey()` and `scikit-image (skimage.restoration.unwrap_phase())`. (#3306)
- Fix deprecation warnings and warnings in the test suite (#3320)
- Add documentation on how the documentation is updated and the required manual changes for minor and major releases. (#3321)
- Add Google Analytics ID to learn more about documentation usage. (#3322)
- Setup workflow to push development documentation automatically. (#3297)

25.3 2.0 (2023-12-20)

25.3.1 Release Highlights

- Hyperspy has split off some of the file reading/writing and domain specific functionalities into separate libraries!
 - [RosettaSciIO](#): A library for reading and writing scientific data files. See [RosettaSciIO release notes](#) for new features and supported formats.
 - [exSpy](#): A library for EELS and EDS analysis. See [exSpy release notes](#) for new features.
 - [HoloSpy](#): A library for analysis of (off-axis) electron holography data. See [HoloSpy release notes](#) for new features.
- The `markers` API has been refactored
 - Lazy markers are now supported
 - Plotting many markers is now *much* faster
 - Added `Polygons` marker
- The documentation has been restructured and improved!
 - Short example scripts are now included in the documentation

- Improved guides for lazy computing as well as an improved developer guide
- Plotting is easier and more consistent:
 - Added horizontal figure layout choice when using the `ipymp1` backend
 - Changing navigation coordinates using the keyboard arrow-keys has been removed. Use `Ctrl + Arrow` instead.
 - Jump to navigation position using `shift + click` in the navigator figure.
- HyperSpy now works with Pyodide/Jupyterlite, checkout hyperspy.org/jupyterlite-hyperspy/!
- The deprecated API has removed: see the list of API changes and removal in the *sections below*.

25.3.2 New features

- `compute()` will now pass keyword arguments to the dask `dask.array.Array.compute()` method. This enables setting the scheduler and the number of computational workers. (#2971)
- Changes to `plot()`:
 - Added horizontal figure layout choice when using the `ipymp1` backend. The default layout can be set in the plot section of the preferences GUI. (#3140)
- Changes to `find_peaks()`:
 - Lazy signals return lazy peak signals
 - `get_intensity` argument added to get the intensity of the peaks
 - The signal axes are now stored in the `metadata.Peaks.signal_axes` attribute of the peaks' signal. (#3142)
- Change the logging output so that logging messages are not displayed in red, to avoid confusion with errors. (#3173)
- Added `hyperspy.decorators.deprecated` and `hyperspy.decorators.deprecated_argument`:
 - Provide consistent and clean deprecation
 - Added a guide for deprecating code (#3174)
- Add functionality to select navigation position using `shift + click` in the navigator. (#3175)
- Added a `plot_residual` to `plot()`. When `True`, a residual line (Signal - Model) appears in the model figure. (#3186)
- Switch to `matplotlib.axes.Axes.pcolormesh()` for image plots involving non-uniform axes. The following cases are covered: 2D-signal with arbitrary navigation-dimension, 1D-navigation and 1D-signal (linescan). Not covered are 2D-navigation images (still uses sliders). (#3192)
- New `interpolate_on_axis()` method to switch one axis of a signal. The data is interpolated in the process. (#3214)
- Added `plot_roi_map()`. Allows interactively using a set of ROIs to select regions of the signal axes of a signal and visualise how the signal varies in this range spatially. (#3224)

25.3.3 Bug Fixes

- Improve syntax in the *io* module. (#3091)
- Fix behaviour of *DictionaryTreeBrowser* setter with value of dictionary type (#3094)
- Avoid slowing down fitting by optimising attribute access of model. (#3155)
- Fix harmless error message when using multiple *RectangularROI*: check if resizer patches are drawn before removing them. Don't display resizers when adding the widget to the figure (widget in unselected state) for consistency with unselected state (#3222)
- Fix keeping dtype in *rebin()* when the endianness is specified in the dtype (#3237)
- Fix serialization error due to `traits.api.Property` not being serializable if a dtype is specified. See #3261 for more details. (#3262)
- Fix setting bounds for "trf", "dogbox" optimizer (#3244)
- Fix bugs in new marker implementation:
 - Markers str representation fails if the marker isn't added to a signal
 - make *from_signal()* to work with all markers - it was only working with *Points* (#3270)
- Documentation fixes:
 - Fix cross-references in documentation and enable sphinx "nitpicky" when building documentation to check for broken links.
 - Fix using mutable objects as default argument.
 - Change some *Component* attributes to properties in order to include their docstrings in the API reference. (#3273)

25.3.4 Improved Documentation

- Restructure documentation:
 - Improve structure of the API reference
 - Improve introduction and overall structure of documentation
 - Add gallery of examples (#3050)
- Add examples to the gallery to show how to use *SpanROI* and slice signal interactively (#3221)
- Add a section on keeping a clean and sensible commit history to the developer guide. (#3064)
- Replace `sphinx.ext.imgmath` by `sphinx.ext.mathjax` to fix the math rendering in the *ReadTheDocs* build (#3084)
- Fix docstring examples in *BaseSignal* class. Describe how to test docstring examples in developer guide. (#3095)
- Update intersphinx_mapping links of matplotlib, numpy and scipy. (#3218)
- Add examples on creating signal from tabular data or reading from a simple text file (#3246)
- Activate checking of example code in docstring and user guide using `doctest` and fix errors in the code. (#3281)
- Update warning of "beta" state in big data section to be more specific. (#3282)

25.3.5 Enhancements

- Add support for passing `**kwargs` to `plot()` when using heatmap style in `plot_spectra()`. (#3219)
- Add support for pep 660 on editable installs for pyproject.toml based builds of extension (#3252)
- Make HyperSpy compatible with pyodide (hence JupyterLite):
 - Set `numba` and `numexpr` as optional dependencies.
 - Replace `dill` by `cloudpickle`.
 - Fallback to `dask` synchronous scheduler when running on pyodide.
 - Reduce packaging size to less than 1MB.
 - Add packaging test on GitHub CI. (#3255)

25.3.6 API changes

- RosettaSciIO was split out of the [HyperSpy repository](#) on July 23, 2022. The IO-plugins and related functions so far developed in HyperSpy were moved to the [RosettaSciIO repository](#). (#2972)
- Extend the IO functions to accept alias names for format name as defined in RosettaSciIO. (#3009)
- Fix behaviour of `print_current_values()`, `print_current_values()` and `print_known_signal_types()`, which were not printing when running from a script - they were only printing when running in notebook or qtconsole. Now all `print_*` functions behave consistently: they all print the output instead of returning an object (string or html). The `IPython.display.display()` will pick a suitable rendering when running in an “ipython” context, for example notebook, qtconsole. (#3145)
- The markers have been refactored - see the new `markers` API and the [gallery of examples](#) for usage. The new `Markers` uses `matplotlib.collections.Collection`, is faster and more generic than the previous implementation and also supports lazy markers. Markers saved in HyperSpy files (`hspy`, `zspy`) with HyperSpy < 2.0 are converted automatically when loading the file. (#3148)
- For all functions with the `rechunk` parameter, the default has been changed from `True` to `False`. This means HyperSpy will not automatically try to change the chunking for lazy signals. The old behaviour could lead to a reduction in performance when working with large lazy datasets, for example 4D-STEM data. (#3166)
- Renamed `Signal2D.crop_image` to `crop_signal()` (#3197)
- Changes and improvement of the map function:
 - Removes the `parallel` argument
 - Replace the `max_workers` with the `num_workers` argument to be consistent with `dask`
 - Adds more documentation on setting the `dask` backend and how to use multiple cores
 - Adds `navigation_chunk` argument for setting the chunks with a non-lazy signal
 - Fix axes handling when the function to be mapped can be applied to the whole dataset - typically when it has the `axis` or `axes` keyword argument. (#3198)
- Remove `physics_tools` since it is not used and doesn't fit in the scope of HyperSpy. (#3235)
- Improve the readability of the code by replacing the `__call__` method of some objects with the more explicit `_get_current_data`.
 - Rename `__call__` method of `BaseSignal` to `_get_current_data`.
 - Rename `__call__` method of `hyperspy.model.BaseModel` to `_get_current_data`.

- Remove `__call__` method of the `hyperspy.component.Component` class. (#3238)
- Rename `hyperspy.api.datasets` to `hyperspy.api.data` and simplify submodule structure:
 - `hyperspy.api.datasets.artificial_data.get_atomic_resolution_tem_signal2d` is renamed to `hyperspy.api.data.atomic_resolution_image()`
 - `hyperspy.api.datasets.artificial_data.get_luminescence_signal` is renamed to `hyperspy.api.data.luminescence_signal()`
 - `hyperspy.api.datasets.artificial_data.get_wave_image` is renamed to `hyperspy.api.data.wave_image()` (#3253)

25.3.7 API Removal

As the HyperSpy API evolves, some of its parts are occasionally reorganized or removed. When APIs evolve, the old API is deprecated and eventually removed in a major release. The functions and methods removed in HyperSpy 2.0 are listed below along with migration advises:

Axes

- `AxesManager.show` has been removed, use `gui()` instead.
- `AxesManager.set_signal_dimension` has been removed, use `as_signal1D()`, `as_signal2D()` or `transpose()` of the signal instance instead.

Components

- The API of the `Polynomial` has changed (it was deprecated in HyperSpy 1.5). The old API had a single parameter `coefficients`, which has been replaced by `a0`, `a1`, etc.
- The legacy option (introduced in HyperSpy 1.6) for `Arctan` has been removed, use `exspy.components.EELSArctan` to use the old API.
- The legacy option (introduced in HyperSpy 1.6) for `Voigt` has been removed, use `exspy.components.PESVoigt` to use the old API.

Data Visualization

- The `saturated_pixels` keyword argument of `plot()` has been removed, use `vmin` and/or `vmax` instead.
- The `get_complex` property of `hyperspy.drawing.signal1d.Signal1DLine` has been removed.
- The keyword argument `line_style` of `plot_spectra()` has been renamed to `linestyle`.
- Changing navigation coordinates using keyboard Arrow has been removed, use `Ctrl + Arrow` instead.
- The `markers` submodules can not be imported from the `api` anymore, use `hyperspy.api.plot.markers` directly, i.e. `hyperspy.api.plot.markers.Arrows`, instead.
- The creation of markers has changed to use their class name instead of aliases, for example, use `m = hs.plot.markers.Lines` instead of `m = hs.plot.markers.line_segment`.

Loading and Saving data

The following deprecated keyword arguments have been removed during the migration of the IO plugins to the *RosettaSciIO* library:

- The arguments `mmap_dir` and `load_to_memory` of the `load()` function have been removed, use the `lazy` argument instead.
- **Bruker composite file (BCF)**: The 'spectrum' option for the `select_type` parameter was removed. Use 'spectrum_image' instead.
- **Electron Microscopy Dataset (EMD) NCEM**: Using the keyword `dataset_name` was removed, use `dataset_path` instead.
- **NeXus data format**: The `dataset_keys`, `dataset_paths` and `metadata_keys` keywords were removed. Use `dataset_key`, `dataset_path` and `metadata_key` instead.

Machine Learning

- The `polyfit` keyword argument has been removed. Use `var_func` instead.
- The list of possible values for the `algorithm` argument of the `decomposition()` method has been changed according to the following table:

Table 1: Change of the algorithm argument

hyperspy < 2.0	hyperspy >= 2.0
fast_svd	SVD along with the argument <code>svd_solver="randomized"</code>
svd	SVD
fast_mlpc	MLPCA along with the argument <code>svd_solver="randomized"</code>
mlpc	MLPCA
nmf	NMF
RPCA_GoDec	RPCA

- The argument `learning_rate` of the ORPCA algorithm has been renamed to `subspace_learning_rate`.
- The argument `momentum` of the ORPCA algorithm has been renamed to `subspace_momentum`.
- The list of possible values for the `centre` keyword argument of the `decomposition()` method when using the SVD algorithm has been changed according to the following table:

Table 2: Change of the centre argument

hyperspy < 2.0	hyperspy >= 2.0
trials	navigation
variables	signal

- For lazy signals, a possible value of the `algorithm` keyword argument of the `decomposition()` method has been changed from "ONMF" to "ORNMF".
- Setting the `metadata` and `original_metadata` attribute of signals is removed, use the `set_item()` and `add_dictionary()` methods of the `metadata` and `original_metadata` attribute instead.

Model fitting

- The `iterpath` default value has changed from 'flyback' to 'serpentine'.
- Changes in the arguments of the `fit()` and `multifit()` methods:
 - The `fitter` argument has been renamed to `optimizer`.
 - The list of possible values for the `optimizer` argument has been renamed according to the following table:

Table 3: Renaming of the optimizer argument

hyperspy < 2.0	hyperspy >= 2.0
<code>fmin</code>	Nelder-Mead
<code>fmin_cg</code>	CG
<code>fmin_ncg</code>	Newton-CG
<code>fmin_bfgs</code>	Newton-BFGS
<code>fmin_l_bfgs_b</code>	L-BFGS-B
<code>fmin_tnc</code>	TNC
<code>fmin_powell</code>	Powell
<code>mpfit</code>	lm
<code>leastsq</code>	lm

- * `loss_function="ml"` has been renamed to `loss_function="ML-poisson"`.
- * `grad=True` has been changed to `grad="analytical"`.
- * The `ext_bounding` argument has been renamed to `bounded`.
- * The `min_function` argument has been removed, use the `loss_function` argument instead.
- * The `min_function_grad` argument has been removed, use the `grad` argument instead.
- The following *BaseModel* methods have been removed:
 - `hyperspy.model.BaseModel.set_boundaries`
 - `hyperspy.model.BaseModel.set_mpfit_parameters_info`
- The arguments `parallel` and `max_workers` have been removed from the `as_signal()` methods.
- Setting the metadata attribute of a *Samfire* has been removed, use the `set_item()` and `add_dictionary()` methods of the metadata attribute instead.
- The deprecated `twin_function` and `twin_inverse_function` have been privatized.
- Remove fancy argument of `print_current_values()` and `print_current_values()`, which wasn't changing the output rendering.
- The attribute `channel_switches` of *BaseModel* have been privatized, instead use the `set_signal_range_from_mask()` or any other methods to set the signal range, such as `set_signal_range()`, `add_signal_range()` or `remove_signal_range()` and their *Model2D* counterparts.

Signal

- `metadata.Signal.binned` is removed, use the `is_binned` axis attribute instead, e. g. `s.axes_manager[-1].is_binned`.
- Some possible values for the `bins` argument of the `get_histogram()` method have been changed according to the following table:

Table 4: Change of the bins argument

hyperspy < 2.0	hyperspy >= 2.0
scotts	scott
freedman	fd

- The `integrate_in_range` method has been removed, use `SpanROI` followed by `integrate1D()` instead.
- The `progressbar` keyword argument of the `compute()` method has been removed, use `show_progressbar` instead.
- The deprecated `comp_label` argument of the methods `plot_decomposition_loadings()`, `plot_decomposition_factors()`, `plot_bss_loadings()`, `plot_bss_factors()`, `plot_cluster_distances()`, `plot_cluster_labels()` has been removed, use the `title` argument instead.
- The `set_signal_type()` now raises an error when passing `None` to the `signal_type` argument. Use `signal_type=""` instead.
- Passing an “iterating over navigation argument” to the `map()` method is removed, pass a HyperSpy signal with suitable navigation and signal shape instead.

Signal2D

- `find_peaks()` now returns lazy signals in case of lazy input signal.

Preferences

- The `warn_if_guis_are_missing` HyperSpy preferences setting has been removed, as it is not necessary anymore.

25.3.8 Maintenance

- Pin third party GitHub actions and add maintenance guidelines on how to update them (#3027)
- Drop support for python 3.7, update oldest supported dependencies and simplify code accordingly (#3144)
- IPython and IPParallel are now optional dependencies (#3145)
- Fix Numpy 1.25 deprecation: implicit array to scalar conversion in `align2D()` (#3189)
- Replace deprecated `scipy.misc` by `scipy.datasets` in documentation (#3225)
- Fix documentation version switcher (#3228)
- Replace deprecated `scipy.interpolate.interp1d` with `scipy.interpolate.make_interp_spline()` (#3233)
- Add support for python 3.12 (#3256)

- Consolidate package metadata:
 - use `pyproject.toml` only
 - clean up unmaintained packaging files
 - use `setuptools_scm` to define version
 - add python 3.12 to test matrix (#3268)
- Pin `pytest-xdist` to 3.5 as a workaround for test suite failure on Azure Pipeline (#3274)

25.4 1.7.6 (2023-11-17)

25.4.1 Bug Fixes

- Allows for loading of `.hspy` files saved with version 2.0.0 and greater and no unit or name set for some axis. (#3241)

25.4.2 Maintenance

- Backport of 3189: fix Numpy1.25 deprecation: implicate array to scalar conversion in `align2D()` (#3243)
- Pin `pillow` to <10.1 to avoid `imageio` error. (#3251)

25.5 1.7.5 (2023-05-04)

25.5.1 Bug Fixes

- Fix plotting boolean array with `plot_images()` (#3118)
- Fix test with `scipy` 1.11 and update deprecated `scipy.interpolate.interp2d` in the test suite (#3124)
- Use intersphinx links to fix links to scikit-image documentation (#3125)

25.5.2 Enhancements

- Improve performance of `model.multifit` by avoiding `axes.is_binned` repeated evaluation (#3126)

25.5.3 Maintenance

- Simplify release workflow and replace deprecated `actions/create-release` action with `softprops/action-gh-release`. (#3117)
- Add support for python 3.11 (#3134)
- Pin `imageio` to <2.28 (#3138)

25.6 1.7.4 (2023-03-16)

25.6.1 Bug Fixes

- Fixes an array indexing bug when loading a .sur file format spectra series. (#3060)
- Speed up `to_numpy` function to avoid slow down when used repeatedly, typically during fitting (#3109)

25.6.2 Improved Documentation

- Replace `sphinx.ext.imgmath` by `sphinx.ext.mathjax` to fix the math rendering in the *ReadTheDocs* build (#3084)

25.6.3 Enhancements

- Add support for Phenom .elid revision 3 and 4 formats (#3073)

25.6.4 Maintenance

- Add `pooch` as test dependency, as it is required to use `scipy.dataset` in latest `scipy` (1.10) and update plotting test. Fix warning when plotting non-uniform axis (#3079)
- Fix `matplotlib` 3.7 and `scikit-learn` 1.4 deprecations (#3102)
- Add support for new pattern to generate random numbers introduced in `dask` 2023.2.1. Deprecate usage of `numpy.random.RandomState` in favour of `numpy.random.default_rng()`. Bump `scipy` minimum requirement to 1.4.0. (#3103)
- Fix checking links in documentation for domain, which aren't compatible with `sphinx linkcheck` (#3108)

25.7 1.7.3 (2022-10-29)

25.7.1 Bug Fixes

- Fix error when reading `Velox` containing FFT with odd number of pixels (#3040)
- Fix pint Unit for `pint>=0.20` (#3052)

25.7.2 Maintenance

- Fix deprecated import of `scipy ascent` in docstrings and the test suite (#3032)
- Fix error handling when trying to convert a ragged signal to non-ragged for `numpy >=1.24` (#3033)
- Fix getting random state `dask` for `dask >=2022.10.0` (#3049)

25.8 1.7.2 (2022-09-17)

25.8.1 Bug Fixes

- Fix some errors and remove unnecessary code identified by LGTM. (#2977)
- Fix error which occurs when guessing output size in the `map()` function and using dask newer than 2022.7.1 (#2981)
- Fix display of x-ray lines when using log norm and the intensity at the line is 0 (#2995)
- Fix handling constant derivative in `spikes_removal_tool()` (#3005)
- Fix removing horizontal or vertical line widget; regression introduced in hyperspy 1.7.0 (#3008)

25.8.2 Improved Documentation

- Add a note in the user guide to explain that when a file contains several datasets, `load()` returns a list of signals instead of a single signal and that list indexation can be used to access a single signal. (#2975)

25.8.3 Maintenance

- Fix extension test suite CI workflow. Enable workflow manual trigger (#2982)
- Fix deprecation warning and time zone test failing on windows (locale dependent) (#2984)
- Fix external links in the documentation and add CI build to check external links (#3001)
- Fix hyperlink in bibliography (#3015)
- Fix matplotlib SpanSelector import for matplotlib 3.6 (#3016)

25.9 1.7.1 (2022-06-18)

25.9.1 Bug Fixes

- Fixes invalid file chunks when saving some signals to hspy/zspy formats. (#2940)
- Fix issue where a TIFF image from an FEI FIB/SEM navigation camera image would not be read due to missing metadata (#2941)
- Respect `show_progressbar` parameter in `map()` (#2946)
- Fix regression in `set_signal_range()` which was raising an error when used interactively (#2948)
- Fix `SpanROI` regression: the output of `interactive()` was not updated when the ROI was changed. Fix errors with updating limits when plotting empty slice of data. Improve docstrings and test coverage. (#2952)
- Fix stacking signals that contain their variance in metadata. Previously it was raising an error when specifying the stacking axis. (#2954)
- Fix missing API documentation of several signal classes. (#2957)
- Fix two bugs in `decomposition()`:
 - The poisson noise normalization was not applied when giving a `signal_mask`

- An error was raised when applying a `signal_mask` on a signal with signal dimension larger than 1. (#2964)

25.9.2 Improved Documentation

- Fix and complete docstrings of `align2D()` and `estimate_shift2D()`. (#2961)

25.9.3 Maintenance

- Minor refactor of the EELS subshells in the `elements` dictionary. (#2868)
- Fix packaging of test suite and tweak tests to pass on different platform of blas implementation (#2933)

25.10 1.7.0 (2022-04-26)

25.10.1 New features

- Add `filter_zero_loss_peak` argument to the `hyperspy._signals.eels.EELSSpectrum.spikes_removal_tool` method (#1412)
- Add `calibrate()` method to `Signal2D` signal, which allows for interactive calibration (#1791)
- Add `hyperspy._signals.eels.EELSSpectrum.vacuum_mask` method to: `hyperspy._signals.eels.EELSSpectrum` signal (#2183)
- Support for *relative slicing* (#2386)
- Implement non-uniform axes, not all hyperspy functionalities support non-uniform axes, see this [tracking issue](#) for progress. (#2399)
- Add (weighted) *linear least square fitting*. Close #488 and #574. (#2422)
- Support for reading JEOL EDS data (#2488)
- Plot overlaid images - see *plotting several images* (#2599)
- Add initial support for *GPU computation* using `cupy` (#2670)
- Add `height` property to the `Gaussian2D` component (#2688)
- Support for reading and writing TVIPS image stream data (#2780)
- Add in *zspy* format: `hspy` specification with the `zarr` format. Particularly useful to speed up loading and *saving large datasets* by using concurrency. (#2825)
- Support for reading DENSolutions Impulse data (#2828)
- Add lazy loading for JEOL EDS data (#2846)
- Add *html representation* for lazy signals and the `get_chunk_size()` method to get the chunk size of given axes (#2855)
- Add support for Hamamatsu HPD-TA Streak Camera tiff files, with axes and metadata parsing. (#2908)

25.10.2 Bug Fixes

- Signals with 1 value in the signal dimension will now be *BaseSignal* (#2773)
- `exspy.material.density_of_mixture()` now throws a Value error when the density of an element is unknown (#2775)
- Improve error message when performing Cliff-Lorimer quantification with a single line intensity (#2822)
- Fix bug for the hydrogenic gdos k edge (#2859)
- Fix bug in axes.UnitConversion: the offset value was initialized by units. (#2864)
- Fix bug where the `map()` function wasn't operating properly when an iterating signal was larger than the input signal. (#2878)
- In case the Bruker defined XML element node at SpectrumRegion contains no information on the specific selected X-ray line (if there is only single line available), suppose it is 'Ka' line. (#2881)
- When loading Bruker Bcf, `cutoff_at_kV=None` does no cutoff (#2898)
- Fix bug where the `map()` function wasn't operating properly when an iterating signal was not an array. (#2903)
- Fix bug for not saving ragged arrays with dimensions larger than 2 in the ragged dimension. (#2906)
- Fix bug with importing some spectra from eelsdb and add progress bar (#2916)
- Fix bug when the `spikes_removal_tool` would not work interactively for signal with 0-dimension navigation space. (#2918)

25.10.3 Deprecations

- Deprecate `hyperspy.axes.AxesManager.set_signal_dimension` in favour of using `as_signal1D()`, `as_signal2D()` or `transpose()` of the signal instance instead. (#2830)

25.10.4 Enhancements

- *Region of Interest (ROI)* can now be created without specifying values (#2341)
- mpfit cleanup (#2494)
- Document reading Attolight data with the sur/pro format reader (#2559)
- Lazy signals now caches the current data chunk when using multifit and when plotting, improving performance. (#2568)
- Read cathodoluminescence metadata from digital micrograph files, amended in PR #2894 (#2590)
- Add possibility to search/access nested items in DictionaryTreeBrowser (metadata) without providing full path to item. (#2633)
- Improve `map()` function in *BaseSignal* by utilizing dask for both lazy and non-lazy signals. This includes adding a `lazy_output` parameter, meaning non-lazy signals now can output lazy results. See the *user guide* for more information. (#2703)
- NeXus file with more options when reading and writing (#2725)
- Add dtype argument to `rebin()` (#2764)
- Add option to set output size when exporting images (#2791)
- Add `switch_iterpath()` context manager to switch iterpath (#2795)

- Add options not to close file (lazy signal only) and not to write dataset for hspy file format, see [HSPy - HyperSpy's HDF5 Specification](#) for details (#2797)
- Add Github workflow to run test suite of extension from a pull request. (#2824)
- Add *ragged* attribute to *BaseSignal* to clarify when a signal contains a ragged array. Fix inconsistency caused by ragged array and add a *ragged array* section to the user guide (#2842)
- Import hyperspy submodules lazily to speed up importing hyperspy. Fix autocompletion *signals* submodule (#2850)
- Add support for JEOL SigtX tiff file (#2862)
- Add new markers `hyperspy.drawing._markers.arrow`, `hyperspy.drawing._markers.ellipse` and filled `hyperspy.drawing._markers.rectangle`. (#2871)
- Add metadata about the file-reading and saving operations to the Signals produced by `load()` and `save()` (see the *metadata structure* section of the user guide) (#2873)
- expose Stage coordinates and rotation angle in metadata for sem images in bcf reader. (#2911)

25.10.5 API changes

- `metadata.Signal.binned` is replaced by an axis parameter, e. g. `axes_manager[-1].is_binned` (#2652)
 - when loading Bruker bcf, `cutoff_at_kV=None` (default) applies no more automatic cutoff.
 - New acceptable values "zealous" and "auto" do automatic cutoff. (#2910)
- Deprecate the ability to directly set `metadata` and `original_metadata` Signal attributes in favor of using `set_item()` and `add_dictionary()` methods or specifying metadata when creating signals (#2913)

25.10.6 Maintenance

- Fix warning when build doc and formatting user guide (#2762)
- Drop support for python 3.6 (#2839)
- Continuous integration fixes and improvements; Bump minimal version requirement of dask to 2.11.0 and matplotlib to 3.1.3 (#2866)
- Tweak tests tolerance to fix tests failure on aarch64 platform; Add python 3.10 build. (#2914)
- Add support for matplotlib 3.5, simplify maintenance of RangeWidget and some signal tools. (#2922)
- Compress some tiff tests files to reduce package size (#2926)

25.11 1.6.5 (2021-10-28)

25.11.1 Bug Fixes

- Suspend plotting during `exspy.models.EELSModel.smart_fit()` call (#2796)
- make `add_marker()` also check if the plot is not active before plotting signal (#2799)
- Fix irresponsive ROI added to a signal plot with a right hand side axis (#2809)
- Fix `plot_histograms()` drawstyle following matplotlib API change (#2810)
- Fix incorrect `map()` output size of lazy signal when input and output axes do not match (#2837)

- Add support for latest h5py release (3.5) (#2843)

25.11.2 Deprecations

- Rename `line_style` to `linestyle` in `plot_spectra()` to match matplotlib argument name (#2810)

25.11.3 Enhancements

- `add_widget()` can now take a string or integer instead of tuple of string or integer (#2809)

25.12 1.6.4 (2021-07-08)

25.12.1 Bug Fixes

- Fix parsing EELS aperture label with unexpected value, for example ‘Imaging’ instead of ‘5 mm’ (#2772)
- Lazy datasets can now be saved out as blockfiles (blo) (#2774)
- ComplexSignals can now be rebinned without error (#2789)
- Method `estimate_parameters()` of the *Polynomial* component now supports order greater than 10 (#2790)
- Update minimal requirement of dependency `importlib_metadata` from `>= 1.6.0` to `>= 3.6` (#2793)

25.12.2 Enhancements

- When saving a dataset with a dtype other than `uint8` to a blockfile (blo) it is now possible to provide the argument `intensity_scaling` to map the intensity values to the reduced range (#2774)

25.12.3 Maintenance

- Fix image comparison failure with numpy 1.21.0 (#2774)

25.13 1.6.3 (2021-06-10)

25.13.1 Bug Fixes

- Fix ROI snapping regression (#2720)
- Fix `shift1D()`, `align1D()` and `hyperspy._signals.eels.EELSSpectrum.align_zero_loss_peak` regression with navigation dimension larger than one (#2729)
- Fix disconnecting events when closing figure and `remove_background()` is active (#2734)
- Fix `map()` regression of lazy signal with navigation chunks of size of 1 (#2748)
- Fix unclear error message when reading a hspy file saved using blosc compression and `hdf5plugin` hasn’t been imported previously (#2760)
- Fix saving navigator of lazy signal (#2763)

25.13.2 Enhancements

- Use `importlib_metadata` instead of `pkg_resources` for extensions registration to speed up the import process and making it possible to install extensions and use them without restarting the python session (#2709)
- Don't import hyperspy extensions when registering extensions (#2711)
- Improve docstrings of various fitting methods (#2724)
- Improve speed of `shift1D()` (#2750)
- Add support for recent EMPAD file; scanning size wasn't parsed. (#2757)

25.13.3 Maintenance

- Add drone CI to test arm64 platform (#2713)
- Fix latex doc build on github actions (#2714)
- Use towncrier to generate changelog automatically (#2717)
- Fix test suite to support dask 2021.4.1 (#2722)
- Generate changelog when building doc to keep the changelog of the development doc up to date on <https://hyperspy.readthedocs.io/en/latest> (#2758)
- Use mamba and conda-forge channel on azure pipeline (#2759)

25.14 1.6.2 (2021-04-13)

This is a maintenance release that adds support for python 3.9 and includes numerous bug fixes and enhancements. See the [issue tracker](#) for details.

25.14.1 Bug Fixes

- Fix disconnect event when closing navigator only plot (fixes #996), (#2631)
- Fix incorrect chunksize when saving EMD NCEM file and specifying chunks (#2629)
- Fix `find_peaks()` GUIs call with laplacian/difference of gaussian methods (#2622 and #2647)
- Fix various bugs with `CircleWidget` and `Line2DWidget` (#2625)
- Fix setting signal range of model with negative axis scales (#2656)
- Fix and improve mask handling in lazy decomposition; Close #2605 (#2657)
- Plot scalebar when the axis scales have different sign, fixes #2557 (#2657)
- Fix `align1D()` returning zeros shifts (#2675)
- Fix finding dataset path for EMD NCEM file containing more than one dataset in a group (#2673)
- Fix squeeze function for multiple zero-dimensional entries, improved docstring, added to user guide. (#2676)
- Fix error in Cliff-Lorimer quantification using absorption correction (#2681)
- Fix `navigation_mask` bug in decomposition when provided as numpy array (#2679)
- Fix closing image contrast tool and setting `vmin/vmax` values (#2684)

- Fix range widget with matplotlib 3.4 (#2684)
- Fix bug in `interactive()` with function returning `None`. Improve user guide example. (#2686)
- Fix broken events when changing signal type #2683
- Fix setting offset in rebin: the offset was changed in the wrong axis (#2690)
- Fix reading XRF bruker file, close #2689 (#2694)

25.14.2 Enhancements

- Widgets plotting improvement and add `pick_tolerance` to plot preferences (#2615)
- Pass keyword argument to the image IO plugins (#2627)
- Improve error message when file not found (#2597)
- Add update instructions to user guide (#2621)
- Improve plotting navigator of lazy signals, add `navigator` setter to lazy signals (#2631)
- Use 'dask_auto' when `rechunk=True` in `change_dtype()` for lazy signal (#2645)
- Use dask chunking when saving lazy signal instead of rechunking and leave the user to decide what is the suitable chunking (#2629)
- Added lazy reading support for FFT and DPC datasets in FEI emd datasets (#2651).
- Improve error message when initialising SpanROI with `left >= right` (#2604)
- Allow running the test suite without the `pytest-mpl` plugin (#2624)
- Add Releasing guide (#2595)
- Add support for python 3.9, fix deprecation warning with matplotlib 3.4.0 and bump minimum requirement to numpy 1.17.1 and dask 2.1.0. (#2663)
- Use native endianness in numba jitted functions. (#2678)
- Add option not to snap ROI when calling the `interactive()` method of a ROI (#2686)
- Make `DictionaryTreeBrowser` lazy by default - see #368 (#2623)
- Speed up setting CI on azure pipeline (#2694)
- Improve performance issue with the `map` method of lazy signal (#2617)
- Add option to copy/load original metadata in `hs.stack` and `hs.load` to avoid large `original_metadata` which can slowdown processing. Close #1398, #2045, #2536 and #1568. (#2691)

25.14.3 Maintenance

- Fix warnings when building documentation (#2596)
- Drop support for `numpy<1.16`, in line with NEP 29 and fix protochip reader for `numpy 1.20` (#2616)
- Run test suite against upstream dependencies (numpy, scipy, scikit-learn and scikit-image) (#2616)
- Update external links in the loading data section of the user guide (#2627)
- Fix various future and deprecation warnings from numpy and scikit-learn (#2646)
- Fix `iterpath` `VisibleDeprecationWarning` when using `fit_component()` (#2654)
- Add integration test suite documentation in the developer guide. (#2663)

- Fix SkewNormal component compatibility with sympy 1.8 (#2701)

25.15 1.6.1 (2020-11-28)

This is a maintenance release that adds compatibility with h5py 3.0 and includes numerous bug fixes and enhancements. See the [issue tracker](#) for details.

25.16 1.6.0 (2020-08-05)

25.16.1 NEW

- Support for the following file formats:
 - DigitalSurf format (SUR & PRO)
 - Phenom ELID format
 - NeXus data format
 - Universal Spectroscopy and Imaging Data (h5USID)
 - EMPAD format (XML & RAW)
 - Prismatic EMD format, see [Electron Microscopy Dataset \(EMD\)](#)
- `hyperspy._signals.eels.EELSSpectrum.print_edges_near_energy` method that, if the `hyperspy-gui-ipywidgets` package is installed, includes an awesome interactive mode. See [Elemental composition of the sample](#).
- Model asymmetric line shape components:
 - *Doniach*
 - *SplitVoigt*
- EDS absorption correction.
- *Argand diagram for complex signals*.
- *Multiple peak finding algorithms for 2D signals*.
- *Cluster analysis*.

25.16.2 Enhancements

- The `get_histogram()` now uses numpy's `np.histogram_bin_edges()` and supports all of its `bins` keyword values.
- Further improvements to the contrast adjustment tool. Test it by pressing the `h` key on any image.
- The following components have been rewritten using *Expression*, boosting their speeds among other benefits.
 - *Arctan*
 - *Voigt*
 - *HeavisideStep*

- The model fitting `fit()` and `multifit()` methods have been vastly improved. See *Fitting the model to the data* and the API changes section below.
- New serpentine iteration path for multi-dimensional fitting. See *Fitting multidimensional datasets*.
- The `plot_spectra()` function now listens to events to update the figure automatically. See *this example*.
- Improve thread-based parallelism. Add `max_workers` argument to the `map()` method, such that the user can directly control how many threads they launch.
- Many improvements to the `decomposition()` and `blind_source_separation()` methods, including support for scikit-learn like algorithms, better API and much improved documentation. See *Machine learning* and the API changes section below.
- Add option to calculate the absolute thickness to the EELS `hyperspy._signals.eels.EELSSpectrum.estimate_thickness` method. See *Thickness estimation*.
- Vastly improved performance and memory footprint of the `estimate_shift2D()` method.
- The `remove_background()` method can now remove Doniach, exponential, Lorentzian, skew normal, split Voigt and Voigt functions. Furthermore, it can return the background model that includes an estimation of the reduced chi-squared.
- The performance of the maximum-likelihood PCA method was greatly improved.
- All ROIs now have a `__getitem__` method, enabling e.g. using them with the `unpack *` operator. See *Slicing using ROIs* for an example.
- New syntax to set the contrast when plotting images. In particular, the `vmin` and `vmax` keywords now take values like `vmin="30th"` to clip the minimum value to the 30th percentile. See *Fast Fourier Transform (FFT)* for an example.
- The `plot()` and `plot()` methods take a new keyword argument `autoscale`. See *Customising image plot* for details.
- The contrast editor and the decomposition methods can now operate on complex signals.
- The default colormap can now be set in *preferences*.

25.16.3 API changes

- The `plot()` keyword argument `saturated_pixels` is deprecated. Please use `vmin` and/or `vmax` instead.
- The `load()` keyword argument `dataset_name` has been renamed to `dataset_path`.
- The `set_signal_type()` method no longer takes `None`. Use the empty string `""` instead.
- The `get_histogram()` `bins` keyword values have been renamed as follows for consistency with numpy:
 - `"scotts"` -> `"scott"`,
 - `"freedman"` -> `"fd"`
- Multiple changes to the syntax of the `fit()` and `multifit()` methods:
 - The `fitter` keyword has been renamed to `optimizer`.
 - The values that the `optimizer` keyword take have been renamed for consistency with scipy:
 - * `"fmin"` -> `"Nelder-Mead"`,
 - * `"fmin_cg"` -> `"CG"`,
 - * `"fmin_ncg"` -> `"Newton-CG"`,
 - * `"fmin_bfgs"` -> `"BFGS"`,

- * "fmin_l_bfgs_b" -> "L-BFGS-B",
 - * "fmin_tnc" -> "TNC",
 - * "fmin_powell" -> "Powell",
 - * "mpfit" -> "lm" (in combination with "bounded=True"),
 - * "leastsq" -> "lm",
- Passing integer arguments to `parallel` to select the number of workers is now deprecated. Use `parallel=True`, `max_workers={value}` instead.
 - The `method` keyword has been renamed to `loss_function`.
 - The `loss_function` value "ml" has been renamed to "ML-poisson".
 - The `grad` keyword no longer takes boolean values. It takes the following values instead: "fd", "analytical", callable or None.
 - The `ext_bounding` keyword has been deprecated and will be removed. Use `bounded=True` instead.
 - The `min_function` keyword argument has been deprecated and will be removed. Use `loss_function` instead.
 - The `min_function_grad` keyword arguments has been deprecated and will be removed. Use `grad` instead.
 - The `iterpath` default will change from 'flyback' to 'serpentine' in HyperSpy version 2.0.
- The following *BaseModel* methods are now private:
 - `hyperspy.model.BaseModel.set_boundaries`
 - `hyperspy.model.BaseModel.set_mpfit_parameters_info`
 - The `comp_label` keyword of the machine learning plotting functions has been renamed to `title`.
 - The *orpc*a constructor's `learning_rate` keyword has been renamed to `subspace_learning_rate`
 - The *orpc*a constructor's `momentum` keyword has been renamed to `subspace_momentum`
 - The *svd_pca* constructor's `centre` keyword values have been renamed as follows:
 - "trials" -> "navigation"
 - "variables" -> "signal"
 - The `bounds` keyword argument of the *decomposition*() is deprecated and will be removed.
 - Several syntax changes in the *decomposition*() method:
 - Several `algorithm` keyword values have been renamed as follows:
 - * "svd": "SVD",
 - * "fast_svd": "SVD",
 - * "nmf": "NMF",
 - * "fast_mlpca": "MLPCA",
 - * "mlpca": "MLPCA",
 - * "RPCA_GoDec": "RPCA",
 - The `polyfit` argument has been deprecated and will be removed. Use `var_func` instead.

25.17 1.5.2 (2019-09-06)

This is a maintenance release that adds compatibility with Numpy 1.17 and Dask 2.3.0 and fixes a bug in the Bruker reader. See [the issue tracker](#) for details.

25.18 1.5.1 (2019-07-28)

This is a maintenance release that fixes some regressions introduced in v1.5. Follow the following links for details on all the [bugs fixed](#).

25.19 1.5.0 (2019-07-27)

25.19.1 NEW

- New method `hyperspy.component.Component.print_current_values()`. See *the User Guide* for details.
- New `hyperspy._components.skew_normal.SkewNormal` component.
- New `hyperspy.api.signals.BaseSignal.apply_apodization()` method and `apodization` keyword for `hyperspy.api.signals.BaseSignal.fft()`. See *Fast Fourier Transform (FFT)* for details.
- Estimation of number of significant components by the elbow method. See *Scree plots*.

25.19.2 Enhancements

- The contrast adjustment tool has been hugely improved. Test it by pressing the `h` key on any image.
- The *Developer Guide* has been extended, enhanced and divided into chapters.
- Signals with signal dimension equal to 0 and navigation dimension 1 or 2 are automatically transposed when using `hyperspy.api.plot.plot_images()` or `hyperspy.api.plot.plot_spectra()` respectively. This is specially relevant when plotting the result of EDS quantification. See *Energy-Dispersive X-ray Spectrometry (EDS)* for examples.
- The following components have been rewritten using `hyperspy._components.expression.Expression`, boosting their speeds among other benefits. Multiple issues have been fixed on the way.
 - `hyperspy._components.lorentzian.Lorentzian`
 - `hyperspy._components.exponential.Exponential`
 - `hyperspy._components.bleasdale.Bleasdale`
 - `hyperspy._components.rc.RC`
 - `hyperspy._components.logistic.Logistic`
 - `hyperspy._components.error_function.Erf`
 - `hyperspy._components.gaussian2d.Gaussian2D`
 - `exspy.components.VolumePlasmonDrude`
 - `exspy.components.DoublePowerLaw`

- The `hyperspy._components.polynomial_deprecated.Polynomial` component will be deprecated in HyperSpy 2.0 in favour of the new `hyperspy._components.polynomial.Polynomial` component, that is based on `hyperspy._components.expression.Expression` and has an improved API. To start using the new component pass the `legacy=False` keyword to the `hyperspy._components.polynomial_deprecated.Polynomial` component constructor.

25.19.3 For developers

- Drop support for python 3.5
- New extension mechanism that enables external packages to register HyperSpy objects. See *Writing packages that extend HyperSpy* for details.

25.20 1.4.2 (2019-06-19)

This is a maintenance release. Among many other fixes and enhancements, this release fixes compatibility issues with Matplotlib v 3.1. Follow the following links for details on all the [bugs fixed](#) and [enhancements](#).

25.21 1.4.1 (2018-10-23)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#) and [enhancements](#).

This release fixes compatibility issues with Python 3.7.

25.22 1.4.0 (2018-09-02)

This is a minor release. Follow the following links for details on all the [bugs fixed](#), [enhancements](#) and [new features](#).

25.22.1 NEW

- Support for three new file formats:
 - Reading FEI’s Velox EMD file format based on the HDF5 open standard. See [EMD \(Velox\)](#).
 - Reading Bruker’s SPX format. See [Bruker](#).
 - Reading and writing the mrcz open format. See [MRCZ format](#).
- New `hyperspy.datasets.artificial_data` module which contains functions for generating artificial data, for use in things like docstrings or for people to test HyperSpy functionalities. See [Loading example data and data from online databases](#).
- New `fft()` and `ifft()` signal methods. See [Fast Fourier Transform \(FFT\)](#).
- New `holospy.signals.HologramImage.statistics()` method to compute useful hologram parameters. See [Further processing of complex wave and phase](#).
- Automatic axes units conversion and better units handling using `pint`. See [Using quantity and converting units](#).
- New `Line2DROI.angle()` method. See [Region Of Interest \(ROI\)](#) for details.

25.22.2 Enhancements

- `plot_images()` improvements (see *Plotting several images* for details):
 - The `cmap` option of `plot_images()` supports iterable types, allowing the user to specify different colormaps for the different images that are plotted by providing a list or other generator.
 - Clicking on an individual image updates it.
- New customizable keyboard shortcuts to navigate multi-dimensional datasets. See *Data visualization*.
- The `remove_background()` method now operates much faster in multi-dimensional datasets and adds the options to interactively plot the remainder of the operation and to set the removed background to zero. See *Background removal* for details.
- The `plot()` method now takes a `norm` keyword that can be “linear”, “log”, “auto” or a matplotlib norm. See *Customising image plot* for details. Moreover, there are three new extra keyword arguments, `fft_shift` and `power_spectrum`, that are useful when plotting fourier transforms. See *Fast Fourier Transform (FFT)*.
- The `align2D()` and `estimate_shift2D()` can operate with sub-pixel accuracy using skimage’s upsampled matrix-multiplication DFT. See *Signal registration and alignment*.

25.23 1.3.2 (2018-07-03)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#) and [enhancements](#).

25.24 1.3.1 (2018-04-19)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#) and [enhancements](#).

Starting with this version, the HyperSpy WinPython Bundle distribution is no longer released in sync with HyperSpy. For HyperSpy WinPython Bundle releases see <https://github.com/hyperspy/hyperspy-bundle>

25.25 1.3.0 (2017-05-27)

This is a minor release. Follow the following links for details on all the [bugs fixed](#), [feature](#) and [documentation enhancements](#), and [new features](#).

25.25.1 NEW

- `rebin()` supports upscaling and rebinning to arbitrary sizes through linear interpolation. See *Rebinning*. It also runs faster if `numba` is installed.
- `signal_extent` and `navigation_extent` properties to easily get the extent of each space.
- New IPywidgets Graphical User Interface (GUI) elements for the Jupyter Notebook. See the new `hyperspy_gui_ipywidgets` package. It is not installed by default, see *Installing HyperSpy* for details.
- All the *Region Of Interest (ROI)* now have a `gui` method to display a GUI if at least one of HyperSpy’s GUI packages are installed.

25.25.2 Enhancements

- Creating many markers is now much faster.
- New “Stage” metadata node. See *Metadata structure* for details.
- The Bruker file reader now supports the new version of the format. See *Bruker*.
- HyperSpy is now compatible with all matplotlib backends, including the nbagg which is particularly convenient for interactive data analysis in the *Jupyter Notebook* in combination with the new *hyperspy_gui_ipywidgets* package. See *Starting Python in Windows*.
- The `vmin` and `vmax` arguments of the `plot_images()` function now accept lists to enable setting these parameters for each plot individually.
- The `plot_decomposition_results()` and `plot_bss_results()` methods now makes a better guess of the number of navigators (if any) required to visualise the components. (Previously they were always plotting four figures by default.)
- All functions that take a signal range can now take a *SpanROI*.
- The following ROIs can now be used for indexing or slicing (see *here* for details):
 - *Point1DROI*
 - *Point2DROI*
 - *SpanROI*
 - *RectangularROI*

25.25.3 API changes

- Permanent markers (if any) are now displayed when plotting by default.
- HyperSpy no longer depends on traitsui (fixing many installation issues) and ipywidgets as the GUI elements based on these packages have now been splitted into separate packages and are not installed by default.
- The following methods now raise a `ValueError` when not providing the number of components if `output_dimension` was not specified when performing a decomposition. (Previously they would plot as many figures as available components, usually resulting in memory saturation):
 - `plot_decomposition_results()`.
 - `plot_decomposition_factors()`.
- The default extension when saving to HDF5 following HyperSpy’s specification is now `hspy` instead of `hdf5`. See *HSpy - HyperSpy’s HDF5 Specification*.
- The following methods are deprecated and will be removed in HyperSpy 2.0
 - `.axes.AxesManager.show`. Use `gui()` instead.
 - All `notebook_interaction` method. Use the equivalent `gui` method instead.
 - `hyperspy.api.signals.Signal1D.integrate_in_range`. Use `integrate1D()` instead.
- The following items have been removed from *preferences*:
 - `General.default_export_format`
 - `General.lazy`
 - `Model.default_fitter`

- `Machine_learning.multiple_files`
 - `Machine_learning.same_window`
 - `Plot.default_style_to_compare_spectra`
 - `Plot.plot_on_load`
 - `Plot.pylab_inline`
 - `EELS.fine_structure_width`
 - `EELS.fine_structure_active`
 - `EELS.fine_structure_smoothing`
 - `EELS.synchronize_cl_with_ll`
 - `EELS.preedge_safe_window_width`
 - `EELS.min_distance_between_edges_for_fine_structure`
- New `Preferences.GUIs` section to enable/disable the installed GUI toolkits.

25.25.4 For developers

- In addition to adding ipywidgets GUI elements, the traitsui GUI elements have been splitted into a separate package. See the new `hyperspy_gui_traitsui` package.
- The new `hyperspy.ui_registry` enables easy connection of external GUI elements to HyperSpy. This is the mechanism used to split the traitsui and ipywidgets GUI elements.

25.26 1.2.0 (2017-02-02)

This is a minor release. Follow the following links for details on all the [bugs fixed](#), [enhancements](#) and [new features](#).

25.26.1 NEW

- Lazy loading and evaluation. See *Working with big data*.
- Parallel `map()` and all the functions that use it internally (a good fraction of HyperSpy's functionaly). See *Iterating external functions with the map method*.
- [Electron Holography](#) reconstruction.
- Support for reading [EDAX TEAM/Genesis \(SPC, SPD\)](#) files.
- New signal methods `indexmin()` and `valuemin()`.

25.26.2 Enhancements

- Easier creation of *Expression* components using substitutions. See the *User Guide for details*.
- *Expression* takes two dimensional functions that can automatically include a rotation parameter. See the *User Guide for details*.
- Better support for EMD files.
- The scree plot got a beauty treatment and some extra features. See *Scree plots*.
- *map()* can now take functions that return differently-shaped arrays or arbitrary objects, see *Iterating external functions with the map method*.
- Add support for stacking multi-signal files. See *Loading multiple files*.
- Markers can now be saved to hdf5 and creating many markers is easier and faster. See *Markers*.
- Add option to save to HDF5 file using the “.hspy” extension instead of “.hdf5”. See *HSPy - HyperSpy’s HDF5 Specification*. This will be the default extension in HyperSpy 1.3.

25.26.3 For developers

- Most of HyperSpy plotting features are now covered by unittests. See *Plot testing*.
- unittests migrated from nose to pytest. See *Running and writing tests*.

25.27 1.1.2 (2079-01-12)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#) and [enhancements](#).

25.28 1.1.1 (2016-08-24)

This is a maintenance release. Follow the following link for details on all the [bugs fixed](#).

25.28.1 Enhancements

- Prettier X-ray lines labels.
- New metadata added to the HyperSpy metadata specifications: `magnification`, `frame_number`, `camera_length`, `authors`, `doi`, `notes` and `quantity`. See *Metadata structure* for details.
- The y-axis label (for 1D signals) and colorbar label (for 2D signals) are now taken from the new metadata. `Signal.quantity`.
- The time and date metadata are now stored in the ISO 8601 format.
- All metadata in the HyperSpy metadata specification is now read from all supported file formats when available.

25.29 1.1.0 (2016-08-03)

This is a minor release. Follow the following links for details on all the [bugs fixed](#).

25.29.1 NEW

- *Transposing (changing signal spaces)*.
- Protochips logfile reader.

25.29.2 Enhancements

- *fit()* takes a new algorithm, the global optimizer *differential evolution*.
- *fit()* algorithm, *leastsq*, inherits SciPy's bound constraints support (requires SciPy >= 0.17).
- *fit()* algorithm names changed to be consistent *scipy.optimize.minimize()* notation.

25.30 1.0.1 (2016-07-27)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#).

25.31 1.0.0 (2016-07-14)

This is a major release. Here we only list the highlist. A detailed list of changes is [available in github](#).

25.31.1 NEW

- *Region Of Interest (ROI)*.
- *Robust PCA* (RPCA) and online RPCA algorithms.
- Numpy ufuncs can now *operate on HyperSpy's signals*.
- ComplexSignal and specialised subclasses to *operate on complex data*.
- Events *logging*.
- Query and fetch spectr from `exspy.data.eelsdb()` from The EELS Database.
- *Interactive Operations*.
- *Events*.

Model

- *Smart Adaptive Multi-dimensional Fitting (SAMFire)*.
- Store *models* in *hdf5* files.
- Add *fancy indexing* to *Model*.
- :ref:Two-dimensional model fitting (*Model2D*).

EDS

- Z-factors quantification.
- Cross section quantification.
- EDS curve fitting.
- X-ray absorption coefficient database.

IO

- Support for reading certain files without *loading them to memory*.
- Bruker's composite file (bcf) reading support.
- Electron Microscopy Datasets (EMD) read and write support.
- SEMPER unf read and write support.
- DENS heat log read support.
- NanoMegas blockfile read and write support.

25.31.2 Enhancements

- More useful AxesManager repr string with html repr for Jupyter Notebook.
- Better progress bar (tqdm).
- Add support for *writing/reading scale and unit* to *tif* files to be read with ImageJ or DigitalMicrograph.

25.31.3 Documentation

- The following sections of the User Guide were revised and largely overwritten:
 - *Installing HyperSpy*.
 - *Machine learning*.
 - Energy-Dispersive X-ray Spectrometry (EDS).
- New *Introduction*.

25.31.4 API changes

- Split *components* into `components1D` and `components2D`.
- Remove `record_by` from metadata.
- Remove simulation classes.
- The *Signal1D*, `hyperspy._signals.image.Signal2D` and *BaseSignal* classes deprecated the old *Spectrum Image* and *Signal* classes.

25.32 0.8.5 (2016-07-02)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#), [feature](#) and [documentation enhancements](#).

It also includes a new feature and introduces an important API change that will be fully enforced in Hyperspy 1.0.

25.32.1 New feature

- Widgets to interact with the model components in the Jupyter Notebook. See [here](#) and [#1007](#).

25.32.2 API changes

The new *BaseSignal*, *Signal1D* and *Signal2D* deprecate `hyperspy.signal.Signal`, *Signal1D* and *Signal2D* respectively. Also `as_signal1D`, `as_signal2D``, `to_signal1D` and `to_signal2D` deprecate `as_signal1D`, `as_signal2D`, `to_spectrum` and `to_image`. See [#963](#) and [#943](#) for details.

25.33 0.8.4 (2016-03-04)

This release adds support for Python 3 and drops support for Python 2. In all other respects it is identical to 0.8.3.

25.34 0.8.3 (2016-03-04)

This is a maintenance release that includes fixes for multiple bugs, some enhancements, new features and API changes. This is set to be the last HyperSpy release for Python 2. The release (HyperSpy 0.8.4) will support only Python 3.

Importantly, the way to start HyperSpy changes (again) in this release. Please read carefully [Starting Python in Windows](#) for details.

The broadcasting rules have also changed. See [Signal operations](#) for details.

Follow the following links for details on all the [bugs fixed](#), [documentation enhancements](#), [enhancements](#), [new features](#) and [API changes](#)

25.35 0.8.2 (2015-08-13)

This is a maintenance release that fixes an issue with the Python installers. Those who have successfully installed 0.8.1 do not need to upgrade.

25.36 0.8.1 (2015-08-12)

This is a maintenance release. Follow the following links for details on all the [bugs fixed](#), [feature](#) and [documentation](#) enhancements.

Importantly, the way to start HyperSpy changes in this release. Read *Starting Python in Windows* for details.

It also includes some new features and introduces important API changes that will be fully enforced in Hyperspy 1.0.

25.36.1 New features

- Support for IPython 3.0.
- `%hyperspy` IPython magic to easily and transparently import HyperSpy, matplotlib and numpy when using IPython.
- *Expression* model component to easily create analytical function components. More details [here](#).
- `hyperspy.signal.Signal.unfolded` context manager.
- `hyperspy.signal.Signal.derivative` method.
- *syntax to access the components in the model* that includes pretty printing of the components.

25.36.2 API changes

- `hyperspy.hspy` is now deprecated in favour of the new *hyperspy.api*. The new API renames and/or move several modules as follows:

- `hspy.components` -> `hyperspy.api.model.components`
- `hspy.utils` -> `hyperspy.api`
- `hspy.utils.markers` `hyperspy.api.plot.markers`
- `hspy.utils.example_signals` -> `hyperspy.api.datasets.example_signals`

In HyperSpy 0.8.1 the full content of `hyperspy.hspy` is still imported in the user namespace, but this can now be disabled in `hs.preferences.General.import_hspy`. In Hyperspy 1.0 it will be disabled by default and the `hyperspy.hspy` module will be fully removed in HyperSpy 0.10. We encourage all users to migrate to the new syntax. For more details see *Starting Python in Windows*.

- Indexing the `hyperspy.signal.Signal` class is now deprecated. We encourage all users to use `isig` and `inav` instead for indexing.
- `hyperspy.hspy.create_model` is now deprecated in favour of the new equivalent `hyperspy.signal.Signal.create_model` `Signal` method.
- `hyperspy.signal.Signal.unfold_if_multidim` is deprecated.

25.37 0.8.0 (2015-04-07)

25.37.1 New features

Core

- `spikes_removal_tool()` displays derivative max value when used with GUI.
- Progress-bar can now be suppressed by passing `show_progressbar` argument to all functions that generate it.

IO

- HDF5 file format now supports saving lists, tuples, binary strings and signals in metadata.

Plotting

- New class, `hyperspy.drawing.marker.MarkerBase`, to plot markers with `hspy.utils.plot.markers` module. See *Markers*.
- New method to plot images with the `plot_images()` function in `hspy.utils.plot.plot_images`. See *Plotting several images*.
- Improved `hyperspy._signals.image.Signal2D.plot` method to customize the image. See *Customising image plot*.

EDS

- New method for quantifying EDS TEM spectra using Cliff-Lorimer method, `hyperspy._signals.eds_tem.EDSTEMSpectrum.quantification`. See *EDS Quantification*.
- New method to estimate for background subtraction, `hyperspy._signals.eds.EDSSpectrum.estimate_background_windows`. See *Background subtraction*.
- New method to estimate the windows of integration, `hyperspy._signals.eds.EDSSpectrum.estimate_integration_windows`.
- New specific `hyperspy._signals.eds.EDSSpectrum.plot` method, with markers to indicate the X-ray lines, the window of integration or/and the windows for background subtraction. See *Plotting X-ray lines*.
- New examples of signal in the `hspy.utils.example_signals` module.
 - `hyperspy.misc.example_signals_loading.load_1D_EDS_SEM_spectrum`
 - `hyperspy.misc.example_signals_loading.load_1D_EDS_TEM_spectrum`
- New method to mask the vacuum, `hyperspy._signals.eds_tem.EDSTEMSpectrum.vacuum_mask` and a specific `hyperspy._signals.eds_tem.EDSTEMSpectrum.decomposition` method that incorporate the vacuum mask

25.37.2 API changes

- *Component* and *Parameter* now inherit `traits.api.HasTraits` that enable `traitsui` to modify these objects.
- `hyperspy.misc.utils.attrsetter` is added, behaving as the default python `setattr` with nested attributes.
- **Several widget functions were made internal and/or renamed:**
 - `add_patch_to` -> `_add_patch_to`
 - `set_patch` -> `_set_patch`
 - `onmove` -> `_onmousemove`
 - `update_patch_position` -> `_update_patch_position`
 - `update_patch_size` -> `_update_patch_size`
 - `add_axes` -> `set_mpl_ax`

25.38 0.7.3 (2015-08-22)

This is a maintenance release. A list of fixed issues is available in the [0.7.3 milestone](#) in the github repository.

25.39 0.7.2 (2015-08-22)

This is a maintenance release. A list of fixed issues is available in the [0.7.2 milestone](#) in the github repository.

25.40 0.7.1 (2015-06-17)

This is a maintenance release. A list of fixed issues is available in the [0.7.1 milestone](#) in the github repository.

25.40.1 New features

- Add suspend/resume model plot updating. See *Visualizing the model*.

25.41 0.7.0 (2014-04-03)

25.41.1 New features

Core

- New syntax to index the *AxesManager*.
- New Signal methods to transform between Signal subclasses. More information [here](#).
 - `hyperspy.signal.Signal.set_signal_type`
 - `hyperspy.signal.Signal.set_signal_origin`
 - `hyperspy.signal.Signal.as_signal2D`

– `hyperspy.signal.Signal.as_signal1D`

- The string representation of the `Signal` class now prints the shape of the data and includes a separator between the navigation and the signal axes e.g (100, 10| 5) for a signal with two navigation axes of size 100 and 10 and one signal axis of size 5.
- Add support for RGBA data. See *Changing the data type*.
- The default toolkit can now be saved in the preferences.
- Added full compatibility with the Qt toolkit that is now the default.
- Added compatibility with the the GTK and TK toolkits, although with no GUI features.
- It is now possible to run HyperSpy in a headless system.
- Added a CLI to `hyperspy.signal.Signal1DTools.remove_background`.
- New `hyperspy.signal.Signal1DTools.estimate_peak_width` method to estimate peak width.
- New methods to integrate over one axis: `hyperspy.signal.Signal.integrate1D` and `hyperspy.signal.Signal1DTools.integrate_in_range`.
- New `hyperspy.signal.Signal.metadata` attribute, `Signal.binned`. Several methods behave differently on binned and unbinned signals. See *Binned and unbinned signals*.
- New `hyperspy.signal.Signal.map` method to easily transform the data using a function that operates on individual signals. See *Iterating over the navigation axes*.
- New `hyperspy.signal.Signal.get_histogram` and `hyperspy.signal.Signal.print_summary_statistics` methods.
- The spikes removal tool has been moved to the *Signal1D* class so that it is available for all its subclasses.
- The `hyperspy.signal.Signal.split`` method now can automatically split back stacked signals into its original part. See *Splitting and stacking*.

IO

- Improved support for FEI's emi and ser files.
- Improved support for Gatan's dm3 files.
- Add support for reading Gatan's dm4 files.

Plotting

- Use the blitting capabilities of the different toolkits to speed up the plotting of images.
- Added several extra options to the `Signal` `hyperspy.signal.Signal.plot` method to customize the navigator. See *Data visualization*.
- Add compatibility with IPython's matplotlib inline plotting.
- New function, `plot_spectra()`, to plot several spectra in the same figure. See *Plotting several spectra*.
- New function, `plot_signals()`, to plot several signals at the same time. See *Plotting several signals*.
- New function, `plot_histograms()`, to plot the histograms of several signals at the same time. See *Plotting several signals*.

Curve fitting

- The chi-squared, reduced chi-squared and the degrees of freedom are computed automatically when fitting. See [Fitting the model to the data](#).
- New functionality to plot the individual components of a model. See [Visualizing the model](#).
- New method, `hyperspy.model.Model.fit_component`, to help setting the starting parameters. See [Setting the initial parameters](#).

Machine learning

- The PCA scree plot can now be easily obtained as a Signal. See [Scree plots](#).
- The decomposition and blind source separation components can now be obtained as `hyperspy.signal.Signal` instances. See [Clustering plots](#).
- New methods to plot the decomposition and blind source separation results that support n-dimensional loadings. See [Visualizing results](#).

Dielectric function

- New `hyperspy.signal.Signal` subclass, `hyperspy._signals.dielectric_function.DielectricFunction`.

EELS

- New method, `hyperspy._signals.eels.EELSSpectrum.kramers_kronig_analysis` to calculate the dielectric function from low-loss electron energy-loss spectra based on the Kramers-Kronig relations. See [Kramers-Kronig Analysis](#).
- New method to align the zero-loss peak, `hyperspy._signals.eels.EELSSpectrum.align_zero_loss_peak`.

EDS

- New signal, `EDSSpectrum` specialized in EDS data analysis, with subsignal for EDS with SEM and with TEM: `EDSSEMpectrum` and `EDSTEMSpectrum`. See [Energy-Dispersive X-ray Spectrometry \(EDS\)](#).
- New database of EDS lines available in the `elements` attribute of the `hspy.utils.material` module.
- Adapted methods to calibrate the spectrum, the detector and the microscope. See [Microscope and detector parameters](#).
- Specific methods to describe the sample, `hyperspy._signals.eds.EDSSpectrum.add_elements` and `hyperspy._signals.eds.EDSSpectrum.add_lines`. See [Describing the sample](#)
- New method to get the intensity of specific X-ray lines: `hyperspy._signals.eds.EDSSpectrum.get_lines_intensity`. See [Plotting](#)

25.41.2 API changes

- hyperspy.misc has been reorganized. Most of the functions in misc.utils has been relocated to specialized modules. misc.utils is no longer imported in hyperspy.hspy. A new hyperspy.utils module is imported instead.
- Objects that have been renamed
 - hspy.elements -> utils.material.elements.
 - Signal.navigation_indexer -> inav.
 - Signal.signal_indexer -> isig.
 - Signal.mapped_parameters -> Signal.metadata.
 - Signal.original_parameters -> Signal.original_metadata.
- The metadata has been reorganized. See [Metadata structure](#).
- The following signal methods now operate out-of-place:
 - hyperspy.signal.Signal.swap_axes
 - hyperspy.signal.Signal.rebin

25.42 0.6.0 (2013-05-25)

25.42.1 New features

- Signal now supports indexing and slicing. See [Indexing](#).
- Most arithmetic and rich arithmetic operators work with signal. See [Signal operations](#).
- Much improved EELSSpectrum methods: hyperspy._signals.eels.EELSSpectrum.estimate_zero_loss_peak_centre, hyperspy._signals.eels.EELSSpectrum.estimate_elastic_scattering_intensity and hyperspy._signals.eels.EELSSpectrum.estimate_elastic_scattering_threshold.
- The axes can now be given using their name e.g. s.crop("x", 1,10)
- New syntax to specify position over axes: an integer specifies the indexes over the axis and a floating number specifies the position in the axis units e.g. s.crop("x", 1, 10.) crops over the axis x (in meters) from index 1 to value 10 meters. Note that this may make your old scripts behave in unexpected ways as just renaming the old *_in_units and *_in_values methods won't work in most cases.
- Most methods now use the natural order i.e. X,Y,Z.. to index the axes.
- Add padding to fourier-log and fourier-ratio deconvolution to fix the wrap-around problem and increase its performance.
- New hyperspy.components.eels_cl_edge.EELSCLEdge.get_fine_structure_as_spectrum EELSCLEdge method.
- New hyperspy.components.arctan.Arctan model component.
- New hyperspy.model.Model.enable_adjust_position and hyperspy.model.Model.disable_adjust_position to easily change the position of components using the mouse on the plot.
- New Model methods hyperspy.model.Model.set_parameters_value, hyperspy.model.Model.set_parameters_free and hyperspy.model.Model.set_parameters_not_free to easily set several important component attributes of a list of components at once.

- New `stack()` function to stack signals.
- New Signal methods: `hyperspy.signal.Signal.integrate_simpson`, `hyperspy.signal.Signal.max`, `hyperspy.signal.Signal.min`, `hyperspy.signal.Signal.var`, and `hyperspy.signal.Signal.std`.
- New sliders window to easily navigate signals with `navigation_dimension > 2`.
- The Ripple (rpl) reader can now read rpl files produced by INCA.

25.42.2 API changes

- The following functions has been renamed or removed:
 - `components.EELSCLEdge`
 - * `knots_factor` -> `fine_structure_smoothing`
 - * `edge_position` -> `onset_energy`
 - * `energy_shift` removed
 - `components.Voigt.origin` -> `centre`
 - `signals.Signal1D`
 - * `find_peaks_1D` -> `Signal.find_peaks1D_ohaver`
 - * `align_1D` -> `Signal.align1D`
 - * `shift_1D` -> `Signal.shift1D`
 - * `interpolate_1D` -> `Signal.interpolate1D`
 - `signals.Signal2D.estimate_2D_translation` -> `Signal.estimate_shift2D`
 - `Signal`
 - * `split_in` -> `split`
 - * `crop_in_units` -> `crop`
 - * `crop_in_pixels` -> `crop`
- Change syntax to create Signal objects. Instead of a dictionary `Signal.__init__` takes keywords e.g with a new syntax `>>> s = signals.Signal1D(np.arange(10))` instead of `>>> s = signals.Signal1D({'data': np.arange(10)})`

25.43 0.5.1 (2012-09-28)

25.43.1 New features

- New Signal method `get_current_signal` proposed by magnunor.
- New Signal `save` method keyword `extension` to easily change the saving format while keeping the same file name.
- New EELSSpectrum methods: `estimate_elastic_scattering_intensity`, `fourier_ratio_deconvolution`, `richardson_lucy_deconvolution`, `power_law_extrapolation`.
- New Signal1D method: `hanning_taper`.

25.43.2 Major bugs fixed

- The `print_current_values` Model method was raising errors when fine structure was enabled or when `only_free = False`.
- The `load` function `signal_type` keyword was not passed to the readers.
- The spikes removal tool was unable to find the next spikes when the spike was detected close to the limits of the spectrum.
- `load` was raising an `UnicodeError` when the title contained non-ASCII characters.
- In Windows `HyperSpy Here` was opening in the current folder, not in the selected folder.
- The fine structure coefficients were overwritten with their std when charging values from the model.
- Storing the parameters in the maps and all the related functionality was broken for 1D spectrum.
- `Remove_background` was broken for 1D spectrum.

25.43.3 API changes

- `EELSSpectrum.find_low_loss_centre` was renamed to `estimate_zero_loss_peak_centre`.
- `EELSSpectrum.calculate_FWHM` was renamed to `estimate_FWHM`.

25.44 0.5.0 (2012-09-07)

25.44.1 New features

- The documentation was thoroughly revised, courtesy of M. Walls.
- New user interface to remove spikes from EELS spectra.
- New `align2D` signals.Signal2D method to align image stacks.
- When loading image files, the data are now automatically converted to grayscale when all the color channels are equal.
- Add the possibility to load a stack memory mapped (similar to ImageJ virtual stack).
- Improved `hyperspy` starter script that now includes the possibility to start HyperSpy in the new IPython notebook.
- Add “HyperSpy notebook here” to the Windows context menu.
- The information displayed in the plots produced by `Signal.plot` have been enhanced.
- Added Egerton’s `sigmak3` and `sigma3` GOS calculations (translated from matlab by I. Iyengar) to the EELS core loss component.
- A browsable dictionary containing the chemical elements and their onset energies is now available in the user namespace under the variable name `elements`.
- The ripple file format now supports storing the beam energy, the collection and the convergence angle.

25.44.2 Major bugs fixed

- The EELS core loss component had a bug in the calculation of the relativistic gamma that produced a gamma that was always approximately zero. As a consequence the GOS calculation was wrong, especially for high beam energies.
- Loading msa files was broken when running on Python 2.7.2 and newer.
- Saving images to rpl format was broken.
- Performing BSS on data decomposed with poissonian noise normalization was failing when some columns or rows of the unfolded data were zero, what occurs often in EDX data for example.
- Importing some versions of scikits learn was broken
- The progress bar was not working properly in the new IPython notebook.
- The contrast of the image was not automatically updated.

25.44.3 API changes

- `spatial_mask` was renamed to `navigation_mask`.
- `Signal1D` and `Signal2D` are not loaded into the user namespace by default. The `signals` module is loaded instead.
- Change the default BSS algorithm to `sklearn.fastica`, that is now distributed with HyperSpy and used in case that `sklearn` is not installed e.g. when using `EPDFree`.
- `_slicing_axes` was renamed to `signal_axes`.
- `_non_slicing_axes` to `navigation_axes`.
- All the `Model *_in_pixels` methods were renamed to `to *_in_pixel`.
- `EELSCLEdge.fs_state` was renamed to `fine_structure_active`.
- `EELSCLEdge.fslist` was renamed to `fine_structure_coeff`.
- `EELSCLEdge.fs_emax` was renamed to `fine_structure_width`.
- `EELSCLEdge.freedelta` was renamed to `free_energy_shift`.
- `EELSCLEdge.delta` was renamed to `energy_shift`.
- A value of `True` in a mask now means that the item is masked all over HyperSpy.

25.45 0.4.1 (2012-04-16)

25.45.1 New features

- Added TIFF 16, 32 and 64 bits support by using (and distributing) Christoph Gohlke's [tiff file library](#).
- Improved UTF8 support.
- Reduce the number of required libraries by making `mdp` and `hdf5` not mandatory.
- Improve the information returned by `__repr__` of several objects.
- `DictionaryBrowser` now has an `export` method, i.e. `mapped_parameters` and `original_parameters` can be exported.
- New `_id_name` attribute for `Components` and `Parameters`. Improvements in their `__repr__` methods.
- `Component.name` can now be overwritten by the user.

- New Signal.__str__ method.
- Include HyperSpy in The Python Package Index.

25.45.2 Bugs fixed

- Non-ascii characters breaking IO and print features fixed.
- Loading of multiple files at once using wildcards fixed.
- Remove broken hyperspy-gui script.
- Remove unmaintained and broken 2D peak finding and analysis features.

25.45.3 Syntax changes

- In EELS automatic background feature creates a PowerLaw component, adds it to the model and add it to a variable in the user namespace. The variable has been renamed from *bg* to *background*.
- pes_gaussian Component renamed to pes_core_line_shape.

25.46 0.4.0 (2012-02-29)

25.46.1 New features

- Add a slider to the filter ui.
- Add auto_replot to sum.
- Add butterworth filter.
- Added centring and auto_transpose to the svd_pca algorithm.
- Keep the mva_results information when changing the signal type.
- Added sparse_pca and mini_batch_sparse_pca to decomposition algorithms.
- Added TV to the smoothing algorithms available in BSS.
- Added whitening to the mdp ICA preprocessing.
- Add explained_variance_ratio.
- Improvements in saving/loading mva data.
- Add option to perform ICA on the scores.
- Add orthomax FA algorithm.
- Add plot methods to Component and Parameter.
- Add plot_results to Model.
- Add possibility to export the decomposition and bss results to a folder.
- Add Signal method *change_dtype*.
- Add the possibility to pass extra parameters to the ICA algorithm.
- Add the possibility to reproject the data after a decomposition.
- Add warning when decomposing a non-float signal.

- adds a method to get the PCs as a Signal1D object and adds smoothing to the ICA preprocessing.
- Add the possibility to select the energy range in which to perform spike removal operations.
- the smoothings guis now offer differentiation and line color option. Smoothing now does not require a gui.
- Fix reverse_ic which was not reversing the scores and improve the autoreversing method.
- Avoid cropping when is not needed.
- Changed criteria to reverse the ICs.
- Changed nonans default to False for plotting.
- Change the whitening algorithm to a svd based one and add sklearn fastica algorithm.
- Clean the ummixing info after a new decomposition.
- Increase the chances that similar independent components will have the same indexes.
- Make savitzky-golay smoothing work without raising figures.
- Make plot_decomposition* plot only the number of factors/scores determined by output_dimension.
- make the Parameter __repr__ method print its name.
- New contrast adjustment tool.
- New export method for Model, Component and Parameter.
- New Model method: print_current_values.
- New signal, spectrum_simulation.
- New smoothing algorithm: total variance denoising.
- Plotting the components in the same or separate windows is now configurable in the preferences.
- Plotting the spikes is now optional.
- Return an error message when the decomposition algorithm is not recognised.
- Store the masks in mva_results.
- The free parameters are now automatically updated on changing the free attribute.

25.46.2 Bugs fixed

- Added missing keywords to plot_pca_factors and plot_ica_factors.
- renamed incorrectly named exportPca and exportIca functions.
- an error was raised when calling generate_data_from_model.
- a signal with containing nans was failing to plot.
- attempting to use any decomposition plotting method after loading with mva_results.load was raising an error.
- a typo was causing in error in pca when normalize_variance = True.
- a typo was raising an error when cropping the decomposition dimension.
- commit 5ff3798105d6 made decomposition and other methods raise an error.
- BUG-FIXED: the decomposition centering index was wrong.
- ensure_directory was failing for the current directory.
- model data forced to be 3D unnecessarily.

- non declared variable was raising an error.
- plot naming for peak char factor plots were messed up.
- plot_RGB was broken.
- plot_scores_2D was using the transpose of the shape to reshape the scores.
- remove background was raising an error when the navigation dimension was 0.
- saving the scores was sometimes transposing the shape.
- selecting indexes while using the learning export functions was raising an error.
- the calibrate ui was calculating wrongly the calibration the first time that Apply was pressed.
- the offset estimation was summing instead of averaging.
- the plot_explained_variance_ratio was actually plotting the cumulative, renamed.
- the signal mask in decomposition and ica was not being raveled.
- the slice attribute was not correctly set at init in some scenarios.
- the smoothing and calibrabron UIs were freezing when the plots where closed before closing the UI window.
- to_spectrum was transposing the navigation dimension.
- variance2one was operating in the wrong axis.
- when closing the plots of a model, the UI object was not being destroyed.
- when plotting an image the title was not displayed.
- when the axis size was changed (e.g. after cropping) the set_signal_dimension method was not being called.
- when using transform the data was being centered and the resulting scores were wrong.

25.46.3 Syntax changes

- in decomposition V rename to explained_variance.
- In FixedPattern, default interpolation changed to linear.
- Line and parable components deleted + improvements in the docstrings.
- pca_V = variance.
- mva_result renamed to learning_results.
- pca renamed to decomposition.
- pca_v and mva_results.v renamed to scores pc renamed to factors . pca_build_SI renamed to get_pca_model ica_build_SI renamed to get_ica_model.
- plot_explained_variance renamed to plot_explained_variance_ratio.
- principal_components_analysis renamed to decomposition.
- rename eels_simulation to eels_spectrum_simulation.
- Rename the output parameter of svd_pca and add scores.
- Replace plot_lev by plot_explained_variance_ratio.
- Scores renamed to loadings.
- slice_bool renamed to navigate to make its function more explicit.

- smoothing renamed to pretreatment and butter added.
- variance2one renamed to normalize_variance.
- w renamed to unmixing matrix and fixes a bug when loading a mva_result in which output_dimension = None.
- ubshells are again available in the interactive session.
- Several changes to the interface.
- The documentation was updated to reflex the last changes.
- The microscopes.csv file was updated so it no longer contains the Orsay VG parameters.

CONTRIBUTE

HyperSpy is a community project maintained for and by its users. There are many ways you can help!

- Help other users on [gitter](#)
- report a bug or request a feature on [GitHub](#)
- or improve the *documentation and code*

26.1 Introduction

This guide is intended to give people who want to start contributing to HyperSpy a foothold to kick-start the process.

We anticipate that many potential contributors and developers will be scientists who may have a lot to offer in terms of expert knowledge but may have little experience when it comes to working on a reasonably large open-source project like HyperSpy. This guide is aimed at you – helping to reduce the barrier to make a contribution.

26.1.1 Getting started

26.1.2 1. Start using HyperSpy and understand it

Probably you would not be interested in contributing to HyperSpy, if you were not already a user, but, just in case: the best way to start understanding how HyperSpy works and to build a broad overview of the code as it stands is to use it – so what are you waiting for? *Install HyperSpy!*

The HyperSpy *User Guide* also provides a good overview of all the parts of the code that are currently implemented as well as much information about how everything works – so read it well.

26.1.3 2. Got a problem? – ask!

Open source projects are all about community – we put in much effort to make good tools available to all and most people are happy to help others start out. Everyone had to start at some point and the philosophy of these projects centres around the fact that we can do better by working together.

Much of the conversation happens in ‘public’ via online platforms. The main two forums used by HyperSpy developers are:

Gitter – where we host a live chat-room in which people can ask questions and discuss things in a relatively informal way.

[Github](#) – the main repository for the source code also enables issues to be raised in a way that means they’re logged until dealt with. This is also a good place to make a proposal for some new feature or tool that you want to work on.

26.1.4 3. Contribute – yes you can!

You don’t need to be a professional programmer to contribute to HyperSpy. Indeed, there are many ways to contribute:

1. Just by asking a question in our [Gitter chat room](#) instead of sending a private email to the developers you are contributing to HyperSpy. Once you get more familiar with HyperSpy, it will be awesome if you could help others with their questions.
2. Issues reported in the [issues tracker](#) are precious contributions.
3. [Pull request](#) reviews are essential for the sustainability of open development software projects and HyperSpy is no exception. Therefore, reviews are highly appreciated. While you may need a good familiarity with the HyperSpy code base to review complex contributions, you can start by reviewing simpler ones such as documentation contributions or simple bug fixes.
4. Last but not least, you can contribute code in the form of documentation, bug fixes, enhancements or new features. That is the main topic of the rest of this guide.

26.1.5 4. Contributing code

You may have a very clear idea of what you want to contribute, but if you’re not sure where to start, you can always look through the issues and pull requests on the [GitHub Page](#). You’ll find that there are many known areas for development in the issues and a number of pull-requests are partially finished projects just sitting there waiting for a keen new contributor to come and learn by finishing.

The documentation (let it be the docstrings, guides or the website) is always in need of some care. Besides, contributing to HyperSpy’s documentation is a very good way to get familiar with GitHub.

When you’ve decided what you’re going to work on – let people know using the online forums! It may be that someone else is doing something similar and can help.; it is also good to make sure that those working on related projects are pulling in the same direction.

There are 3 key points to get right when starting out as a contributor:

1. Work out what you want to contribute and break it down in to manageable chunks. Use [Git branches](#) to keep work separated in manageable sections.
2. Make sure that your [code style](#) is good.
3. Bear in mind that every new function you write will need [tests](#) and [user documentation](#)!

Note: The IO plugins formerly developed within HyperSpy have now been moved to the separate [RosettaSciIO repository](#) in order to facilitate a wider use also by other packages. Plugins supporting additional formats or corrections/enhancements to existing plugins should now be contributed to the [RosettaSciIO repository](#) and file format specific issues should be reported to the [RosettaSciIO issue tracker](#).

26.2 Using Git and GitHub

For developing the code, the home of HyperSpy is on [GitHub](#), and you'll see that a lot of this guide boils down to properly using that platform. So, visit the following link and poke around the code, issues, and pull requests: [HyperSpy on GitHub](#).

It is probably also worth to visit [github.com](#) and to go through the “boot camp” to get a feel for the terminology.

In brief, to give you a hint on the terminology to search for and get accustomed to, the contribution pattern is:

1. Setup git/github, if you don't have it yet.
2. Fork HyperSpy on GitHub.
3. Checkout your fork on your local machine.
4. Create a new branch locally, where you will make your changes.
5. Push the local changes to your own HyperSpy fork on GitHub.
6. Create a pull request (PR) to the official HyperSpy repository.

Note: You cannot mess up the main HyperSpy project unless you have been promoted to write access and the dev-team. So when you're starting out be confident to play, get it wrong, and if it all goes wrong, you can always get a fresh install of HyperSpy!!

PS: If you choose to develop in Windows/Mac you may find [Github Desktop](#) useful.

26.2.1 Use Git and work in manageable branches

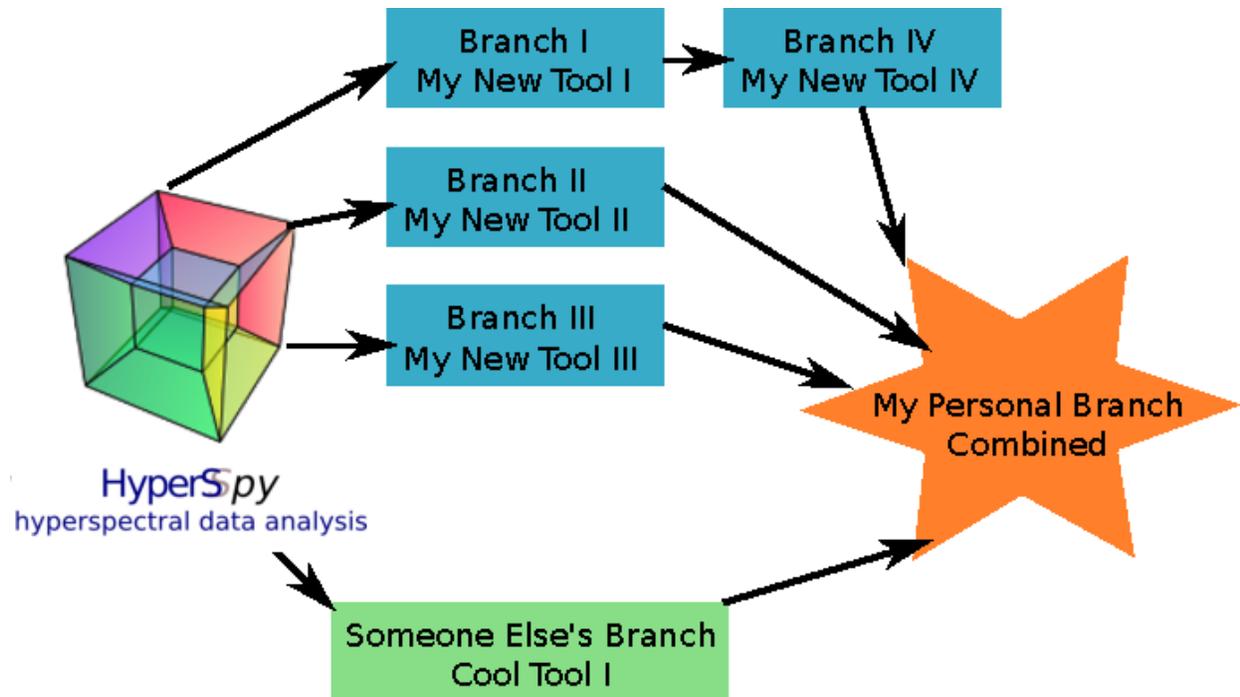
By now, you will have had a look around GitHub – but why is it so important?

Well, GitHub is the public forum in which we manage and discuss development of the code. More importantly, it enables every developer to use *Git*, which is an open source “version control” system. By version control, we mean that you can separate out your contribution to the code into many versions (called branches) and switch between them easily. Later, you can choose which version you want to have integrated into HyperSpy. You can learn all about Git at [git-scm!](#)

It is very important to separate your contributions, so that each branch is a small advancement on the “master” code or on another branch. In the end, each branch will have to be checked and reviewed by someone else before it can be included – so if it is too big, you will be asked to split it up!

For personal use, before integrating things into the main HyperSpy code, you can merge some branches for your personal use. However, make sure each new feature has its own branch that is contributed through a separate pull request!

Diagrammatically, you should be aiming for something like this:



26.2.2 Semantic versioning and HyperSpy main branches

HyperSpy versioning follows [semantic versioning](#) and the version number is therefore a three-part number: MAJOR.MINOR.PATCH. Each number will change depending on the type of changes according to the following:

- MAJOR increases when making incompatible API changes,
- MINOR increases when adding functionality in a backwards compatible manner, and
- PATCH increases when making backwards compatible bug fixes.

The git repository of HyperSpy has 3 main branches matching the above pattern and depending on the type of pull request, you will need to base your pull request on one of the following branch:

- `RELEASE_next_major` to change the API in a not backward-compatible fashion,
- `RELEASE_next_minor` to add new features and improvement,
- `RELEASE_next_patch` for bug fixes.

The `RELEASE_next_patch` branch is merged daily into `RELEASE_next_minor` by the github action [Nightly Merge](#).

26.2.3 Changing base branch

If you started your work in the wrong branch (typically on `RELEASE_next_minor` instead of `RELEASE_next_patch` and you are doing a bug fix), you can change the base branch using `git rebase --onto`, like this:

```
$ git rebase --onto <NEW-BASE-BRANCH> <OLD-BASE-BRANCH> <YOUR-BRANCH>
```

For example, to rebase the `bug_fix_branch` branch from `RELEASE_next_minor` onto `RELEASE_next_patch`:

```
$ git rebase --onto RELEASE_next_patch RELEASE_next_minor bug_fix_branch
```

26.2.4 Keeping the git history clean

For review, and for revisiting changes at a later point, it is advisable to keep a “clean” git history, i.e. a meaningful succession of commits. In some cases, it is useful to rewrite the git history to keep it more readable:

- it is not always possible to keep a clean history and quite often the code development follows an exploratory process with code changes going back and forth, etc.
- Commits that only fix typographic mistakes, formatting or failing tests usually can be *squashed* (merged) into the previous commits.

When using a GUI for interaction with *git*, check out its features for joining and reordering commits.

When using *git* in the command line, use `git rebase` with the *interactive* option. For example, to rearrange the last five commits:

```
$ git rebase -i HEAD~5
```

In a text editor, you can then edit the commit history. If you have commits a . . . e and want to merge b and e into a and d, respectively, while moving c to the end of the history, your file would look the following:

```
pick a ...
squash b ...
pick d ...
squash e ...
pick c ...
```

Afterwards, you get a chance to edit the commit messages.

Finally, to push the changes, use a + in front of the branch name, to override commits you have already pushed to github previously:

```
git push origin +lumberjack-branch
```

See, for example, [How \(and why!\) to keep your Git commit history clean](#) for a more detailed blog post on this subject.

26.3 Running and writing tests

26.3.1 Writing tests

Every new function that is written in to HyperSpy should be tested and documented. HyperSpy uses the `pytest` library for testing. The tests reside in the `hyperspy.tests` module.

Tests are short functions, found in `hyperspy/tests`, that call your functions under some known conditions and check the outputs against known values. They should depend on as few other features as possible so that when they break, we know exactly what caused it. Ideally, the tests should be written at the same time as the code itself, as they are very convenient to run to check outputs when coding. Writing tests can seem laborious but you’ll probably soon find that they are very important, as they force you to sanity-check all the work that you do.

Useful testing hints:

- When comparing integers, it’s fine to use `==`
- When comparing floats, be sure to use `numpy.testing.assert_allclose()` or `numpy.testing.assert_almost_equal()`
- `numpy.testing.assert_allclose()` is also convenient for comparing numpy arrays

- The `hyperspy.misc.test_utils.py` contains a few useful functions for testing
- `@pytest.mark.parametrize()` is a very convenient decorator to test several parameters of the same function without having to write too much repetitive code, which is often error-prone. See [pytest documentation](#) for more details.
- It is good to check that the tests do not use too much memory after creating new tests. If you need to explicitly delete your objects and free memory, you can do the following to release the memory associated with the `s` object:

```
>>> import gc
>>> # Do some work with the object
>>> s = ...
>>> # Clear the memory
>>> del s
>>> gc.collect()
```

26.3.2 Running tests

First ensure `pytest` and its plugins are installed by:

```
# If using a standard hyperspy install
$ pip install hyperspy[tests]

# Or, from a hyperspy local development directory
$ pip install -e .[tests]

# Or just installing the dependencies using conda
$ conda install -c conda-forge pytest pytest-mpl
```

To run them:

```
$ pytest --mpl --pyargs hyperspy
```

Or, from HyperSpy's project folder, simply:

```
$ pytest
```

Note: `pytest` configuration options are set in the `setup.cfg` file, under the `[tool:pytest]` section. See the [pytest configuration documentation](#) for more details.

The HyperSpy test suite can also be run in parallel if you have multiple CPUs available, using the [pytest-xdist](#) plugin. If you have the plugin installed, HyperSpy will automatically run the test suite in parallel on your machine.

```
# To run on all the cores of your machine
$ pytest -n auto --dist loadfile

# To run on 2 cores
$ pytest -n 2 --dist loadfile
```

The `--dist loadfile` argument will group tests by their containing file. The groups are then distributed to available workers as whole units, thus guaranteeing that all tests in a file run in the same worker.

Note: Running tests in parallel using `pytest-xdist` will change the content and format of the output of `pytest` to the console. We recommend installing `pytest-sugar` to produce nicer-looking output including an animated progressbar.

To test docstring examples, assuming the current location is the HyperSpy root directory:

```
# All
$ pytest --doctest-modules --ignore-glob=hyperspy/tests --pyargs hyperspy

# In a single file, like the signal.py file
$ pytest --doctest-modules hyperspy/signal.py
```

26.3.3 Flaky tests

Test functions can sometimes exhibit intermittent or sporadic failure, with seemingly random or non-deterministic behaviour. They may sometimes pass or sometimes fail, and it won't always be clear why. These are usually known as “flaky” tests.

One way to approach flaky tests is to rerun them, to see if the failure was a one-off. This can be achieved using the `pytest-rerunfailures` plugin.

```
# To re-run all test suite failures a maximum of 3 times
$ pytest --reruns 3

# To wait 1 second before the next retry
$ pytest --reruns 3 --reruns-delay 1
```

You can read more about flaky tests in the [pytest documentation](#).

26.3.4 Test coverage

Once you have pushed your pull request to the official HyperSpy repository, you can see the coverage of your tests using the [codecov.io](#) check for your PR. There should be a link to it at the bottom of your PR on the Github PR page. This service can help you to find how well your code is being tested and exactly which parts are not currently tested.

You can also measure code coverage locally. If you have installed `pytest-cov`, you can run (from HyperSpy's project folder):

```
$ pytest --cov=hyperspy
```

Configuration options for code coverage are also set in the `setup.cfg` file, under the `[coverage:run]` and `[coverage:report]` sections. See the [coverage documentation](#) for more details.

Note: The [codecov.io](#) check in your PR will fail if it either decreases the overall test coverage of HyperSpy, or if any of the lines introduced in your diff are not covered.

26.3.5 Continuous integration (CI)

The HyperSpy test suite is run using continuous integration services provided by [Github Actions](#) and [Azure Pipelines](#). In case of Azure Pipelines, CI helper scripts are pulled from the `ci-scripts` repository.

The testing matrix is as follows:

- **Github Actions:** test a range of Python versions on Linux, MacOS and Windows; all dependencies are installed from [PyPI](#). See `.github/workflows/tests.yml` in the HyperSpy repository for further details.
- **Azure Pipeline:** test a range of Python versions on Linux, MacOS and Windows; all dependencies are installed from [Anaconda Cloud](#) using the “conda-forge” channel. See `azure-pipelines.yml` in the HyperSpy repository for further details.
- The testing of **HyperSpy extensions** is described in the *integration test suite* section.

This testing matrix has been designed to be simple and easy to maintain, whilst ensuring that packages from PyPI and Anaconda cloud are not mixed in order to avoid red herring failures of the test suite caused by application binary interface (ABI) incompatibility between dependencies.

The most recent versions of packages are usually available first on PyPI, before they are available on Anaconda Cloud. These means that if a recent release of a dependency breaks the test suite, it should happen first on Github Actions. Similarly, deprecation warnings will usually appear first on Github Actions.

The documentation build is done on both Github Actions and [Read the Docs](#), and it is worth checking that no new warnings have been introduced when writing documentation in the user guide or in the docstrings.

The Github Actions testing matrix also includes the following special cases:

- The test suite is run against HyperSpy’s minimum requirements on Python 3.7 on Linux. This will skip any tests that require **optional** packages such as `scikit-learn`.
- The test suite is run against the oldest supported versions of `numpy`, `matplotlib` and `scipy`. For more details, see this [Github issue](#).
- The test suite is run against the development supported versions of `numpy`, `scipy`, `scikit-learn` and `scikit-image` using the weekly build wheels available on <https://anaconda.org/scipy-wheels-nightly>. For more details, see this [Github issue](#).

26.3.6 Plot testing

Plotting is tested using the `@pytest.mark.mpl_image_compare` decorator of the [pytest mpl plugin](#). This decorator uses reference images to compare with the generated output during the tests. The reference images are located in the folder defined by the argument `baseline_dir` of the `@pytest.mark.mpl_image_compare` decorator.

To run plot tests, you simply need to add the option `--mpl`:

```
$ pytest --mpl
```

If you don’t use `--mpl`, the test functions will be executed, but the images will not be compared to the reference images.

If you need to add or change some plots, follow the workflow below:

1. Write the tests using appropriate decorators such as `@pytest.mark.mpl_image_compare`.
2. If you need to generate a new reference image in the folder `plot_test_dir`, for example, run: `pytest --mpl-generate-path=plot_test_dir`
3. Run again the tests and this time they should pass.
4. Use `git add` to put the new file in the git repository.

When the plotting tests fail, it is possible to download the figure comparison images generated by `pytest-mpl` in the artifacts tabs of the corresponding build on Azure Pipelines:

The screenshot shows the Azure Pipelines build summary for a job triggered by 'ericpre'. The 'Summary' tab is selected. Key information includes:

- Triggered by:** ericpre
- Time started and elapsed:** Sat at 13:48, 18m 56s
- Related:** 0 work items, 3 published; 1 consumed (highlighted with a red box)
- Errors (7):**
 - Bash exited with code '127'. Linux Python38 • Run test suite
 - Bash exited with code '1'. Linux Python37 • Run test suite

The plotting tests are tested on Azure Pipelines against a specific version of `matplotlib` defined in `conda_environment_dev.yml`. This is because small changes in the way `matplotlib` generates the figure between versions can sometimes make the tests fail.

For plotting tests, the `matplotlib` backend is set to `agg` by setting the `MPLBACKEND` environment variable to `agg`. At the first import of `matplotlib.pyplot`, `matplotlib` will look at the `MPLBACKEND` environment variable and accordingly set the backend.

26.3.7 Exporting `pytest` results as HTML

With `pytest-html`, it is possible to export the results of running `pytest` for easier viewing. It can be installed by `conda`:

```
$ conda install pytest-html
```

and run by:

```
$ pytest --mpl --html=report.html
```

See `pytest-mpl` for more details.

26.4 Writing documentation

Documentation comes in two parts: docstrings and user-guide documentation.

26.4.1 Docstrings

Written at the start of a function, they give essential information about how it should be used, such as which arguments can be passed to it and what the syntax should be. The docstrings need to follow the [numpy specification](#), as shown in [this example](#).

As a general rule, any code that is part of the public API (i.e. any function or class that an end-user might access) should have a clear and comprehensive docstring explaining how to use it. Private methods that are never intended to be exposed to the end-user (usually a function or class starting with an underscore) should still be documented to the extent that future developers can understand what the function does.

To test code of “examples” section in the docstring, run:

```
pytest --doctest-modules --ignore=hyperspy/tests
```

You can check whether your docstrings follow the convention by using the `flake8-docstrings` [extension](#), like this:

```
# If not already installed, you need flake8 and flake8-docstrings
pip install flake8 flake8-docstrings

# Run flake8 on your file
flake8 /path/to/your/file.py

# Example output
/path/to/your/file.py:46:1: D103 Missing docstring in public function
/path/to/your/file.py:59:1: D205 1 blank line required between summary line and
↳ description
```

26.4.2 User-guide documentation

A description of the functionality of the code and how to use it with examples and links to the relevant code.

When writing both the docstrings and user guide documentation, it is useful to have some data which the users can use themselves. Artificial datasets for this purpose can be found in [data](#).

Example codes in the user guide can be tested using `doctest`:

```
pytest doc --doctest-modules --doctest-glob="*.rst" -v
```

26.4.3 Build the documentation

To check the output of what you wrote, you can build the documentation by running the `make` command in the `hyperspy/doc` directory. For example `make html` will build the whole documentation in html format. See the `make` command documentation for more details.

To install the documentation dependencies, run either

```
$ conda install hyperspy-dev
```

or

```
$ pip install hyperspy[doc]
```

When writing documentation, the Python package `sphobjinv` can be useful for writing cross-references. For example, to find how to write a cross-reference to `set_signal_type()`, use:

```
$ sphobjinv suggest doc/_build/html/objects.inv set_signal_type -st 90
```

Name	Score

:meth:`hyperspy.signal.BaseSignal.set_signal_type`	90

26.4.4 Hosting versioned documentation

Builds of the documentation for each minor and major release are hosted in the <https://github.com/hyperspy/hyperspy-doc> repository and are used by the `version switcher` of the documentation.

The "dev" version is updated automatically when pushing on the `RELEASE_next_minor` branch and the "current" (stable) version is updated automatically when a tag is pushed. When releasing a minor and major release, two manual steps are required:

1. in <https://github.com/hyperspy/hyperspy-doc>, copy the "current" stable documentation to a separate folder named with the corresponding version
2. update the documentation version switch, in `doc/_static/switcher.json`:
 - copy and paste the "current" documentation entry
 - update the version in the "current" entry to match the version to be released, e.g. increment the minor or major digit
 - in the newly created entry, update the link to the folder created in step 1.

26.5 Coding style

HyperSpy follows the Style Guide for Python Code - these are rules for code consistency that you can read all about in the [Python Style Guide](#). You can use the `black` or `ruff` code formatter to automatically fix the style of your code using pre-commit hooks.

Linting error can be suppressed in the code using the `# noqa` marker, more information in the [ruff documentation](#).

26.6 Pre-commit hooks

Code linting and formatting is checked continuously using `ruff` pre-commit hooks.

These can be run locally by using `pre-commit`. Alternatively, the comment `pre-commit.ci autofix` can be added to a PR to fix the formatting using `pre-commit.ci`.

26.7 Deprecations

HyperSpy follows [semantic versioning](#) where changes follow such that:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backward compatible manner
3. PATCH version when you make backward compatible bug fixes

This means that as little, ideally no, functionality should break between minor releases. Deprecation warnings are raised whenever possible and feasible for functions/methods/properties/arguments, so that users get a heads-up one (minor) release before something is removed or changes, with a possible alternative to be used.

A deprecation decorator should be placed right above the object signature to be deprecated:

```
@deprecated(since=1.7.0, removal=2.0.0, alternative="bar")
def foo(self, n):
    return n + 1

@deprecated_argument(since=1.7.0, removal=2.0.0, name="x", alternative="y")
def this_property(y):
    return y
```

This will update the docstring, and print a visible deprecation warning telling the user to use the alternative function or argument.

These deprecation wrappers are inspired by those in `kikuchipy`.

26.8 Tips for writing methods that work on lazy signals

With the addition of the `LazySignal` class and its derivatives, adding methods that operate on the data becomes slightly more complicated. However, we have attempted to streamline it as much as possible. `LazySignals` use `dask.array.Array` for the data field instead of the usual `numpy.ndarray`. The full documentation is available [here](#). While interfaces of the two arrays are indeed almost identical, the most important differences are (da being `dask.array.Array` in the examples):

- **Dask arrays are immutable:** `da[3] = 2` does not work. `da += 2` does, but it's actually a new object – you might as well use `da = da + 2` for a better distinction.
- **Unknown shapes are problematic:** `res = da[da>0.3]` works, but the shape of the result depends on the values and cannot be inferred without execution. Hence, few operations can be run on `res` lazily, and it should be avoided if possible.
- **Computations in Dask are Lazy:** Dask only preforms a computation when it has to. For example the sum function isn't run until `compute` is called. This also means that some function can be applied to only some portion of the data.

The easiest way to add new methods that work both with arbitrary navigation dimensions and `LazySignals` is by using the `map` method to map your function `func` across all “navigation pixels” (e.g. spectra in a spectrum-image). `map` methods will run the function on all pixels efficiently and put the results back in the correct order. `func` is not constrained by `dask` and can use whatever code (assignment, etc.) you wish.

The `map` function is flexible and should be able to handle most operations that operate on some signal. If you add a `BaseSignal` with the same navigation size as the signal, it will be iterated alongside the mapped signal, otherwise a keyword argument is assumed to be constant and is applied to every signal.

If the new method cannot be coerced into a shape suitable for `map`, separate cases for lazy signals will have to be written. If a function operates on arbitrary-sized arrays and the shape of the output can be known before calling, `da.map_blocks` and `da.map_overlap` are efficient and flexible.

Finally, in addition to `_iterate_signal` that is available to all HyperSpy signals, lazy counterparts also have the `_block_iterator` method that supports signal and navigation masking and yields (returns on subsequent calls) the underlying `dask` blocks as `numpy` arrays. It is important to note that stacking all (flat) blocks and reshaping the result into the initial data shape will not result in identical arrays. For illustration it is best to see the [dask documentation](#).

For a summary of the implementation, see the [first post of the github issue #1219](#).

26.9 Interactive Plotting

Interactive plotting in `hyperspy` is handled through `matplotlib` and is primarily driven through event handling.

Specifically for some signal `s` when the `index` value for some `BaseDataAxis` is changed then the signal plot is updated to reflect the data at that index. Each signal has a `__call__` function which will return the data at the current navigation index.

For lazy signals the `__call__` function works slightly differently as the current chunk is cached. As a result the `__call__` function first checks if the current chunk is cached and then either computes the chunk where the navigation index resides or just pulls the value from the cached chunk.

26.10 Interactive Markers

`class`~.drawing.markers.Markers`` operates in a similar way to signals when the data is retrieved. The current `index` for the signal is used to retrieve the current array of markers at that `index`. Additionally, lazy markers are treated similarly where the current chunk for a marker is cached.

Adding new types of markers to `hyperspy` is relatively simple. Currently `hyperspy` supports any `matplotlib.collections.Collection` object. For most common cases this should be sufficient as `matplotlib` has a large number of built in collections beyond what is available in `hyperspy`.

In the event that you want a specific shape that isn’t supported you can define a custom `matplotlib.path.Path` object and then use the `matplotlib.collections.PathCollection` to add the markers to the plot. Currently there is no support for saving `Path` based markers but that can be added if there are specific use cases.

Many times when annotating 1-D Plots you want to add markers which are relative to the data. For example you may want to add a line which goes from `[0,y]` where `y` is the value at `x`. To do this you can set the `offset_transform` to “relative” or the `transform` to `relative`.

```
>>> s = hs.signals.Signal1D(np.random.rand(3, 15))
>>> from matplotlib.collections import LineCollection
>>> m = hs.plot.markers.Lines(segments=[[2,0],[2,1.0]]), transform = "relative")
>>> s.plot()
>>> s.add_marker(m)
```

This marker will create a line at a value=2 which extends from 0 → 1 and updates as the index changes.

26.11 Speeding up code

Python is not the fastest language, but this is not usually an issue because most scientific Python software uses libraries written in compiled languages such as Numpy for data processing, hence running at close to C-speed. Nevertheless, sometimes it is necessary to improve the speed of some parts of the code by writing some functions in compiled languages or by using Just-in-time (JIT) compilation. Before taking this approach, please make sure that the extra complexity is worth it by writing a first implementation of the functionality using Python and Numpy and profiling your code.

26.11.1 Writing Numba code

If you need to improve the speed of a given part of the code your first choice should be [Numba](#). The motivation is that Numba code is very similar (when not identical) to Python code, and therefore, it is a lot easier to maintain than Cython code (see below).

Numba is also a required dependency for HyperSpy, unlike Cython which is only an optional dependency.

26.11.2 Writing Cython code

Cython code should only be considered if:

1. It is not possible to speed up the function using Numba, and instead,
2. it is accompanied by a pure Python version of the same code that behaves exactly in the same way when the compiled C extension is not present. This extra version is required because we may not be able to provide binaries for all platforms and not all users will be able to compile C code in their platforms.

Please read through the official Cython recommendations (<http://docs.cython.org/>) before writing Cython code.

To help troubleshoot potential deprecations in future Cython releases, add a comment in the header of your .pyx files stating the Cython version you used when writing the code.

Note that the “cythonized” .c or .cpp files are not welcome in the git source repository because they are typically very large.

Once you have written your Cython files, add them to `raw_extensions` in `setup.py`.

Compiling Cython code

If Cython is present in the build environment and any cythonized c/c++ file is missing, then `setup.py` tries to cythonize all extensions automatically.

To make the development easier `setup.py` provides a `recythonize` command that can be used in conjunction with default commands. For example

```
python setup.py recythonize build_ext --inplace
```

will recythonize all Cython code and compile it.

Cythonization and compilation will also take place during continuous integration (CI).

26.12 Writing packages that extend HyperSpy

New in version 1.5: External packages can extend HyperSpy by registering signals, components and widgets.

Warning: The mechanism to register extensions is in beta state. This means that it can change between minor and patch versions. Therefore, if you maintain a package that registers HyperSpy extensions, please verify that it works properly with any future HyperSpy release. We expect it to reach maturity with the release of HyperSpy 2.0.

External packages can extend HyperSpy by registering signals, components and widgets. Objects registered by external packages are “first-class citizens” i.e. they can be used, saved and loaded like any of those objects shipped with HyperSpy. Because of HyperSpy’s structure, we anticipate that most packages registering HyperSpy extensions will provide support for specific sorts of data.

Models can also be provided by external packages, but don’t need to be registered. Instead, they are returned by the `create_model` method of the relevant `BaseSignal` subclass, see for example, the `exspy.signals.EDSTEMSpectrum.create_model()` of the `exspy.signals.EDSTEMSpectrum`.

It is good practice to add all packages that extend HyperSpy to the [list of known extensions](#) regardless of their maturity level. In this way, we can avoid duplication of efforts and issues arising from naming conflicts. This repository also runs an [integration test suite](#) daily, which runs the test suite of all extensions to check the status of the ecosystem. See the [corresponding section](#) for more details.

At this point, it is worth noting that HyperSpy’s main strength is its amazing community of users and developers. We trust that the developers of packages that extend HyperSpy will play by the same rules that have made the Python scientific ecosystem successful. In particular, avoiding duplication of efforts and being good community players by contributing code to the best matching project are essential for the sustainability of our open software ecosystem.

26.12.1 Registering extensions

In order to register HyperSpy extensions, you need to:

1. Add the following line to your package’s `setup.py`:

```
entry_points={'hyperspy.extensions': 'your_package_name =
your_package_name'},
```

2. Create a `hyperspy_extension.yaml` configuration file in your module’s root directory.
3. Declare all new HyperSpy objects provided by your package in the `hyperspy_extension.yaml` file.

For a full example on how to create a package that extends HyperSpy, see [the HyperSpy Sample Extension package](#).

26.12.2 Creating new HyperSpy BaseSignal subclasses

When and where to create a new BaseSignal subclass

HyperSpy provides most of its functionality through the different `BaseSignal` subclasses. A HyperSpy “signal” is a class that contains data for analysis and functions to perform the analysis in the form of class methods. Functions that are useful for the analysis of most datasets are in the `BaseSignal` class. All other functions are in specialized subclasses.

The flowchart below can help you decide where to add a new data analysis function. Notice that only if no suitable package exists for your function, you should consider creating your own.

Registering a new BaseSignal subclass

To register a new *BaseSignal* subclass you must add it to the `hyperspy_extension.yaml` file, as in the following example:

```
signals:
  MySignal:
    signal_type: "MySignal"
    signal_type_aliases:
      - MS
      - ThisIsMySignal
    # The dimension of the signal subspace. For example, 2 for images, 1 for
    # spectra. If the signal can take any signal dimension, set it to -1.
    signal_dimension: 1
    # The data type, "real" or "complex".
    dtype: real
    # True for LazySignal subclasses
    lazy: False
    # The module where the signal is located.
    module: my_package.signal
```

Note that HyperSpy uses `signal_type` to determine which class is the most appropriate to deal with a particular sort of data. Therefore, the signal type must be specific enough for HyperSpy to find a single signal subclass match for each sort of data.

Warning: HyperSpy assumes that only one signal subclass exists for a particular `signal_type`. It is up to external package developers to avoid `signal_type` clashes, typically by collaborating in developing a single package per data type.

The optional `signal_type_aliases` are used to determine the most appropriate signal subclass when using `set_signal_type()`. For example, if the `signal_type` Electron Energy Loss Spectroscopy has an EELS alias, setting the signal type to EELS will correctly assign the signal subclass with Electron Energy Loss Spectroscopy signal type. It is good practice to choose a very explicit `signal_type` while leaving acronyms for `signal_type_aliases`.

26.12.3 Creating new HyperSpy model components

When and where to create a new component

HyperSpy provides the `hyperspy._components.expression.Expression` component that enables easy creation of 1D and 2D components from mathematical expressions. Therefore, strictly speaking, we only need to create new components when they cannot be expressed as simple mathematical equations. However, HyperSpy is all about simplifying the interactive data processing workflow. Therefore, we consider that functions that are commonly used for model fitting, in general or specific domains, are worth adding to HyperSpy itself (if they are of common interest) or to specialized external packages extending HyperSpy.

The flowchart below can help you decide when and where to add a new hyperspy model `hyperspy.component.Component` for your function, should you consider creating your own.

Registering new components

All new components must be a subclass of `hyperspy._components.expression.Expression`. To register a new 1D component add it to the `hyperspy_extension.yaml` file as in the following example:

```
components1D:
# _id_name of the component. It must be a UUID4. This can be generated
# using uuid.uuid4(). Also, many editors can automatically generate
# UUIDs. The same UUID must be stored in the components __id_name__ attribute.
fc731a2c-0a05-4acb-91df-d15743b531c3:
# The module where the component class is located.
module: my_package.components
# The actual class of the component
class: MyComponent1DClass
```

Equivalently, to add a new component 2D:

```
components2D:
# _id_name of the component. It must be a UUID4. This can be generated
# using uuid.uuid4(). Also, many editors can automatically generate
# UUIDs. The same UUID must be stored in the components __id_name__ attribute.
2ffbe0b5-a991-4fc5-a089-d2818a80a7e0:
# The module where the component is located.
module: my_package.components
class: MyComponent2DClass
```

Note: HyperSpy's legacy components use their class name instead of a UUID as `_id_name`. This is for compatibility with old versions of the software. New components (including those provided through the extension mechanism) must use a UUID4 in order to i) avoid name clashes ii) make it easy to find the component online if e.g. the package is renamed or the component relocated.

26.12.4 Creating and registering new widgets and toolkeys

To generate GUIs of specific methods and functions, HyperSpy uses widgets and toolkeys:

- *widgets* (typically ipywidgets or traitsui objects) generate GUIs,
- *toolkeys* are functions which associate widgets to a signal method or to a module function.

An extension can declare new toolkeys and widgets. For example, the `hyperspy-gui-traitsui` and `hyperspy-gui-ipywidgets` provide widgets for toolkeys declared in HyperSpy.

Registering toolkeys

To register a new toolkey:

1. Declare a new toolkey, e. g. by adding the `hyperspy.ui_registry.add_gui_method` decorator to the function you want to assign a widget to.
2. Register a new toolkey that you have declared in your package by adding it to the `hyperspy_extension.yaml` file, as in the following example:

GUI:

```
# In order to assign a widget to a function, that function must declare
# a `toolkey`. The `toolkeys` list contains a list of all the toolkeys
# provided by extensions. In order to avoid name clashes, by convention,
# toolkeys must start with the name of the package that provides them.
```

toolkeys:

```
- my_package.MyComponent
```

Registering widgets

In the example below, we register a new `ipywidget` widget for the `my_package.MyComponent` toolkey of the previous example. The function simply returns the widget to display. The key `module` defines where the functions resides.

GUI:**widgets:****ipywidgets:**

```
# Each widget is declared using a dictionary with two keys, `module` and `function`.
```

my_package.MyComponent:

```
# The function that creates the widget
```

```
function: get_mycomponent_widget
```

```
# The module where the function resides.
```

```
module: my_package.widgets
```

26.12.5 Integration test suite

The `integration test suite` runs the test suite of HyperSpy and of all registered HyperSpy extensions on a daily basis against both the release and development versions. The build matrix is as follows:

Table 1: Build matrix of the integration test suite

HyperSpy	Extension	Dependencies
Release	Release	Release
Release	Development	Release
RELEASE_next_patch	Release	Release
RELEASE_next_patch	Development	Release
RELEASE_next_minor	Release	Release
RELEASE_next_minor	Development	Release
RELEASE_next_minor	Development	Development
RELEASE_next_minor	Development	Pre-release

The development packages of the dependencies are provided by the `scipy-wheels-nightly` repository, which provides `scipy`, `numpy`, `scikit-learn` and `scikit-image` at the time of writing. The pre-release packages are obtained from PyPI and these will be used for any dependency which provides a pre-release package on PyPI.

A similar `Integration test` workflow can run from pull requests (PR) to the `hyperspy` repository when the label `run-extension-tests` is added to a PR or when a PR review is edited.

26.13 Useful information

26.13.1 NEP 29 — Recommend Python and Numpy version support

Abstract

NEP 29 (NumPy Enhancement Proposals) recommends that all projects across the Scientific Python ecosystem adopt a common “time window-based” policy for support of Python and NumPy versions. Standardizing a recommendation for project support of minimum Python and NumPy versions will improve downstream project planning.

Implementation recommendation

This project supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of `numpy` released in the 24 months prior to the project, and at minimum the last three minor versions.

In `setup.py`, the `python_requires` variable should be set to the minimum supported version of Python. All supported minor versions of Python should be in the test matrix and have binary artifacts built for the release.

Minimum Python and NumPy version support should be adjusted upward on every major and minor release, but never on a patch release.

26.13.2 Conda-forge packaging

The feedstock for the conda package lives in the conda-forge organisation on github: [conda-forge/hyperspy-feedstock](https://github.com/conda-forge/hyperspy-feedstock).

26.13.3 Monitoring version distribution

Download metrics are available from pypi and Anaconda cloud, but the reliability of these numbers is poor for the following reason:

- hyperspy is distributed by other means: the [hyperspy-bundle](#), or by various linux distribution (Arch-Linux, openSUSE)
- these packages may be used by continuous integration of other python libraries

However, distribution of downloaded versions can be useful to identify issues, such as version pinning or library incompatibilities. Various services processing the [pypi data](#) are available online:

- pepy.tech
- libraries.io
- pypistats.org

26.13.4 HTML Representations

For use inside of jupyter notebooks, *html* representations are functions which allow for more detailed data representations using snippets of populated HTML.

Hyperspy uses *jinja* and extends *dask's html* representations in many cases in line with this PR: <https://github.com/dask/dask/pull/8019>

26.14 Maintenance

26.14.1 GitHub Workflows

GitHub workflows are used to:

- run the test suite
- build packages and upload to pypi and GitHub release
- build the documentation and check the links (external and cross-references)
- push the development documentation to the dev folder of <https://github.com/hyperspy/hyperspy-doc>

Some of these workflow need to access GitHub “secrets”, which are private to the HyperSpy repository. The personal access token PAT_DOCUMENTATION is set to be able to push the documentation built to the <https://github.com/hyperspy/hyperspy-doc> repository.

To reduce the risk that these “secrets” are made accessible publicly, for example, through the injection of malicious code by third parties in one of the GitHub workflows used in the HyperSpy organisation, the third party actions (those that are not provided by established trusted parties) are pinned to the SHA of a specific commit, which is trusted not to contain malicious code.

26.14.2 Updating GitHub Actions

The workflows in the HyperSpy repository use GitHub actions provided by established trusted parties and third parties. They are updated regularly by the [dependabot](#) in pull requests.

When updating a third party action, the action has to be pinned using the SHA of the commit of the updated version and the corresponding code changes will need to be reviewed to verify that it doesn't include malicious code.

WHAT IS HYPERSPY

HyperSpy is an open source Python library which provides tools to facilitate the interactive data analysis of multidimensional datasets that can be described as multidimensional arrays of a given signal (e.g. a 2D array of spectra a.k.a spectrum image).

HyperSpy aims at making it easy and natural to apply analytical procedures that operate on an individual signal to multidimensional datasets of any size, as well as providing easy access to analytical tools that exploit their multidimensionality.

New in version 1.5: External packages can extend HyperSpy by registering signals, components and widgets.

The functionality of HyperSpy can be extended by external packages, e.g. to implement features for analyzing a particular sort of data (usually related to a specific set of experimental methods). A [list of packages that extend HyperSpy](#) is curated in a dedicated repository. For details on how to register extensions see [Writing packages that extend HyperSpy](#).

Changed in version 2.0: HyperSpy was split into a core package (HyperSpy) that provides the common infrastructure for multidimensional datasets and the dedicated IO package [RosettaSciIO](#). Signal classes focused on specific types of data previously included in HyperSpy (EELS, EDS, Holography) were moved to specialized [HyperSpy extensions](#).

HYPERSPY'S CHARACTER

HyperSpy has been written by a subset of the people who use it, a particularity that sets its character:

- To us, this program is a research tool, much like a screwdriver or a Green's function. We believe that the better our tools are, the better our research will be. We also think that it is beneficial for the advancement of knowledge to share our research tools and to forge them in a collaborative way. This is because by collaborating we advance faster, mainly by avoiding reinventing the wheel. Idealistic as it may sound, many other people think like this and it is thanks to them that this project exists.
- Not surprisingly, we care about making it easy for others to contribute to HyperSpy. In other words, we aim at minimising the “user becomes developer” threshold. Do you want to contribute already? No problem, see the *Introduction* for details.
- The main way of interacting with the program is through scripting. This is because [Jupyter](#) exists, making your interactive data analysis productive, scalable, reproducible and, most importantly, fun. That said, widgets to interact with HyperSpy elements are provided where there is a clear productivity advantage in doing so. See the [hyperspy-gui-ipywidgets](#) and [hyperspy-gui-traitsui](#) packages for details. Not enough? If you need a full, standalone GUI, [HyperSpyUI](#) is for you.

LEARNING RESOURCES

Getting Started

New to HyperSpy or Python? The getting started guide provides an introduction on basic usage of HyperSpy and how to install it.

User Guide

The user guide provides in-depth information on key concepts of HyperSpy and how to use it along with background information and explanations.

Reference

Documentation of the metadata specification and of the Application Programming Interface (API), which describe how HyperSpy functions work and which parameters can be used.

Examples

Gallery of short examples illustrating simple tasks that can be performed with HyperSpy.

Tutorials

Tutorials in form of Jupyter Notebooks to learn how to process multi-dimensional data using HyperSpy.

Contributing

HyperSpy is a community project maintained for and by its users. There are many ways you can help!

CITING HYPERSPY

If HyperSpy has been significant to a project that leads to an academic publication, please acknowledge that fact by citing it. The DOI in the badge below is the [Concept DOI](#) of HyperSpy. It can be used to cite the project without referring to a specific version. If you are citing HyperSpy because you have used it to process data, please use the DOI of the specific version that you have employed. You can find it by clicking on the DOI badge below.

30.1 HyperSpy's citation in the scientific literature

Given the increasing number of articles that cite HyperSpy we do not maintain a list of articles citing HyperSpy. For an up to date list search for HyperSpy in a scientific database e.g. [Google Scholar](#).

Note: Articles published before 2012 may mention the HyperSpy project under its old name, *EELSLab*.

CREDITS

HyperSpy is maintained by [an active community of developers](#).

The HyperSpy logo was created by Stefano Mazzucco. It is a modified version of [this figure](#) and the same GFDL license applies.

BIBLIOGRAPHY

- [1] Matplotlib colors API: https://matplotlib.org/stable/api/colors_api.html.
- [1] V. Satopää, J. Albrecht, D. Irwin, and B. Raghavan. “Finding a “Kneedle” in a Haystack: Detecting Knee Points in System Behavior, . 31st International Conference on Distributed Computing Systems Workshops, pp. 166-171, June 2011.
- [Andrews1997] Darren T. Andrews and Peter D. Wentzell, “Applications of maximum likelihood principal component analysis: incomplete data sets and calibration transfer”, *Analytica Chimica Acta* 350, no. 3 (September 19, 1997): 341-352.
- [Zhao2016] Zhao, Renbo, and Vincent YF Tan. “Online nonnegative matrix factorization with outliers.” *Acoustics, Speech and Signal Processing (ICASSP)*, 2016 IEEE International Conference on. IEEE, 2016.
- [Feng2013] Jiashi Feng, Huan Xu and Shuicheng Yuan, “Online Robust PCA via Stochastic Optimization”, *Advances in Neural Information Processing Systems* 26, (2013), pp. 404-412.
- [Ruder2016] Sebastian Ruder, “An overview of gradient descent optimization algorithms”, arXiv:1609.04747, (2016), <https://arxiv.org/abs/1609.04747>.
- [Zhou2011] Tianyi Zhou and Dacheng Tao, “GoDec: Randomized Low-rank & Sparse Matrix Decomposition in Noisy Case”, *ICML-11*, (2011), pp. 33-40.
- [Kessy2015] A. Kessy, A. Lewin, and K. Strimmer, “Optimal Whitening and Decorrelation”, arXiv:1512.00809, (2015), <https://arxiv.org/pdf/1512.00809.pdf>
- [KeenanKotula2004] M. Keenan and P. Kotula, “Accounting for Poisson noise in the multivariate analysis of ToF-SIMS spectrum images”, *Surf. Interface Anal* 36(3) (2004): 203-212.

PYTHON MODULE INDEX

h

- hyperspy.api, 326
- hyperspy.api.data, 334
- hyperspy.api.model, 336
- hyperspy.api.model.components1D, 337
- hyperspy.api.model.components2D, 361
- hyperspy.api.plot, 375
- hyperspy.api.plot.markers, 362
- hyperspy.api.roi, 382
- hyperspy.api.samfire, 393
- hyperspy.api.samfire.fit_tests, 388
- hyperspy.api.samfire.global_strategies, 389
- hyperspy.api.samfire.local_strategies, 390
- hyperspy.api.signals, 395
- hyperspy.axes, 485
- hyperspy.events, 497
- hyperspy.learn.ml pca, 501
- hyperspy.learn.mva, 500
- hyperspy.learn.ornmf, 502
- hyperspy.learn.orthomax, 505
- hyperspy.learn.rpca, 505
- hyperspy.learn.svd_pca, 509
- hyperspy.learn.whitening, 511
- hyperspy.models.model1d, 526
- hyperspy.models.model2d, 531
- hyperspy.roi, 562
- hyperspy.utils.peakfinders2D, 569

A

- active_components (*hyperspy.model.BaseModel* property), 513
- active_is_multidimensional (*hyperspy.component.Component* property), 533
- add() (*hyperspy.events.EventSuppressor* method), 499
- add_dictionary() (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 565
- add_gaussian_noise() (*hyperspy.api.signals.BaseSignal* method), 397
- add_items() (*hyperspy.api.plot.markers.Markers* method), 367
- add_jobs() (*hyperspy.api.samfire.SamfirePool* method), 393
- add_marker() (*hyperspy.api.signals.BaseSignal* method), 397
- add_node() (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 565
- add_phase_ramp() (*hyperspy.api.signals.ComplexSignal2D* method), 461
- add_poissonian_noise() (*hyperspy.api.signals.BaseSignal* method), 398
- add_ramp() (*hyperspy.api.signals.Signal2D* method), 477
- add_signal_range() (*hyperspy.models.model1d.Model1D* method), 527
- add_signal_range() (*hyperspy.models.model2d.Model2D* method), 531
- add_widget() (*hyperspy.roi.BaseInteractiveROI* method), 562
- AIC_test (*class in hyperspy.api.samfire.fit_tests*), 388
- AICc_test (*class in hyperspy.api.samfire.fit_tests*), 388
- align1D() (*hyperspy.api.signals.Signal1D* method), 465
- align2D() (*hyperspy.api.signals.Signal2D* method), 478
- amplitude (*hyperspy.api.signals.ComplexSignal* property), 457
- angle() (*hyperspy.api.roi.Line2DROI* method), 383
- angle() (*hyperspy.api.signals.ComplexSignal* method), 457
- append() (*hyperspy.model.BaseModel* method), 513
- append() (*hyperspy.models.model1d.Model1D* method), 528
- append() (*hyperspy.samfire.Samfire* method), 540
- apply_apodization() (*hyperspy.api.signals.BaseSignal* method), 399
- Arctan (*class in hyperspy.api.model.components1D*), 338
- argand_diagram() (*hyperspy.api.signals.ComplexSignal* method), 457
- Arrows (*class in hyperspy.api.plot.markers*), 363
- as_dictionary() (*hyperspy.component.Component* method), 533
- as_dictionary() (*hyperspy.component.Parameter* method), 536
- as_dictionary() (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 566
- as_dictionary() (*hyperspy.model.BaseModel* method), 513
- as_lazy() (*hyperspy.api.signals.BaseSignal* method), 399
- as_signal() (*hyperspy.component.Parameter* method), 537
- as_signal() (*hyperspy.model.BaseModel* method), 514
- as_signal1D() (*hyperspy.api.signals.BaseSignal* method), 400
- as_signal2D() (*hyperspy.api.signals.BaseSignal* method), 400
- assign_current_value_to_all() (*hyperspy.component.Parameter* method), 537
- assign_current_values_to_all() (*hyperspy.model.BaseModel* method), 515
- atomic_resolution_image() (*in module hyperspy.api.data*), 334
- axes_are_aligned_with_data (*hyperspy.axes.AxesManager* property), 487
- AxesManager (*class in hyperspy.axes*), 485

B

backup() (*hyperspy.samfire.Samfire* method), 540
 BaseDataAxis (*class in hyperspy.axes*), 490
 BaseInteractiveROI (*class in hyperspy.roi*), 562
 BaseModel (*class in hyperspy.model*), 512
 BaseROI (*class in hyperspy.roi*), 564
 BaseSignal (*class in hyperspy.api.signals*), 396
 BIC_test (*class in hyperspy.api.samfire.fit_tests*), 388
 Bleasdale (*class in hyperspy.api.model.components1D*), 338
 blind_source_separation() (*hyperspy.api.signals.BaseSignal* method), 401

C

calibrate() (*hyperspy.api.signals.Signal1D* method), 466
 calibrate() (*hyperspy.api.signals.Signal2D* method), 478
 change_dtype() (*hyperspy._signals.lazy.LazySignal* method), 548
 change_dtype() (*hyperspy.api.signals.BaseSignal* method), 403
 change_dtype() (*hyperspy.api.signals.ComplexSignal* method), 457
 change_strategy() (*hyperspy.samfire.Samfire* method), 540
 chisq (*hyperspy.model.BaseModel* property), 515
 CircleROI (*class in hyperspy.api.roi*), 383
 Circles (*class in hyperspy.api.plot.markers*), 363
 clean() (*hyperspy.api.samfire.global_strategies.GlobalStrategy* method), 389
 clean() (*hyperspy.api.samfire.global_strategies.HistogramStrategy* method), 389
 clean() (*hyperspy.api.samfire.local_strategies.LocalStrategy* method), 390
 clean() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy* method), 391
 clean_peaks() (*in module hyperspy.utils.peakfinders2D*), 569
 close() (*hyperspy.api.plot.markers.Markers* method), 368
 close_file() (*hyperspy._signals.lazy.LazySignal* method), 549
 cluster_analysis() (*hyperspy.api.signals.BaseSignal* method), 403
 collect_results() (*hyperspy.api.samfire.SamfirePool* method), 394
 CommonSignal1D (*class in hyperspy._signals.common_signal1d*), 545
 CommonSignal2D (*class in hyperspy._signals.common_signal2d*), 545
 compile_function() (*hyperspy.api.model.components1D.Expression* method), 344

ComplexSignal (*class in hyperspy.api.signals*), 456
 ComplexSignal1D (*class in hyperspy.api.signals*), 460
 ComplexSignal2D (*class in hyperspy.api.signals*), 461
 Component (*class in hyperspy.component*), 532
 components (*hyperspy.model.BaseModel* property), 515
 compute() (*hyperspy._signals.lazy.LazySignal* method), 549
 compute_navigator() (*hyperspy._signals.lazy.LazySignal* method), 550
 connect() (*hyperspy.events.Event* method), 497
 connected (*hyperspy.events.Event* property), 497
 convert_to_non_uniform_axis() (*hyperspy.axes.FunctionalDataAxis* method), 494
 convert_to_uniform_axis() (*hyperspy.axes.BaseDataAxis* method), 490
 convert_to_units() (*hyperspy.axes.UnitConversion* method), 496
 convert_units() (*hyperspy.axes.AxesManager* method), 487
 coordinates (*hyperspy.axes.AxesManager* property), 488
 copy() (*hyperspy.api.signals.BaseSignal* method), 405
 copy() (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 566
 create_axes() (*hyperspy.axes.AxesManager* method), 488
 create_model() (*hyperspy.api.signals.Signal1D* method), 466
 create_model() (*hyperspy.api.signals.Signal2D* method), 479
 create_samfire() (*hyperspy.model.BaseModel* method), 515
 crop() (*hyperspy.api.signals.BaseSignal* method), 405
 crop() (*hyperspy.axes.DataAxis* method), 492
 crop() (*hyperspy.axes.FunctionalDataAxis* method), 494
 crop() (*hyperspy.axes.UniformDataAxis* method), 495
 crop_decomposition_dimension() (*hyperspy.learn.mva.LearningResults* method), 500
 crop_signal() (*hyperspy.api.signals.Signal1D* method), 466
 crop_signal() (*hyperspy.api.signals.Signal2D* method), 479

D

data (*hyperspy.api.signals.BaseSignal* property), 405
 DataAxis (*class in hyperspy.axes*), 492
 decomposition() (*hyperspy._signals.lazy.LazySignal* method), 551
 decomposition() (*hyperspy.api.signals.BaseSignal* method), 405
 deepcopy() (*hyperspy.api.signals.BaseSignal* method), 407

- deepcopy() (*hyperspy.misc.utils.DictionaryTreeBrowser method*), 566
- default_traits_view() (*hyperspy.component.Parameter method*), 537
- derivative() (*hyperspy.api.signals.BaseSignal method*), 407
- DictionaryTreeBrowser (*class in hyperspy.misc.utils*), 565
- diff() (*hyperspy._signals.lazy.LazySignal method*), 552
- diff() (*hyperspy.api.signals.BaseSignal method*), 408
- disable_adjust_position() (*hyperspy.models.model1d.Model1D method*), 528
- disable_plot_components() (*hyperspy.model.BaseModel method*), 515
- disable_plot_components() (*hyperspy.models.model1d.Model1D method*), 528
- disconnect() (*hyperspy.events.Event method*), 497
- dof (*hyperspy.model.BaseModel property*), 515
- Doniach (*class in hyperspy.api.model.components1D*), 339
- ## E
- Ellipses (*class in hyperspy.api.plot.markers*), 364
- enable_adjust_position() (*hyperspy.models.model1d.Model1D method*), 528
- enable_plot_components() (*hyperspy.model.BaseModel method*), 515
- enable_plot_components() (*hyperspy.models.model1d.Model1D method*), 528
- ensure_parameters_in_bounds() (*hyperspy.model.BaseModel method*), 516
- Erf (*class in hyperspy.api.model.components1D*), 340
- estimate_elbow_position() (*hyperspy.api.signals.BaseSignal method*), 409
- estimate_number_of_clusters() (*hyperspy.api.signals.BaseSignal method*), 410
- estimate_parameters() (*hyperspy.api.model.components1D.Doniach method*), 340
- estimate_parameters() (*hyperspy.api.model.components1D.Exponential method*), 342
- estimate_parameters() (*hyperspy.api.model.components1D.Gaussian method*), 345
- estimate_parameters() (*hyperspy.api.model.components1D.GaussianHF method*), 346
- estimate_parameters() (*hyperspy.api.model.components1D.Lorentzian method*), 349
- estimate_parameters() (*hyperspy.api.model.components1D.Offset method*), 350
- estimate_parameters() (*hyperspy.api.model.components1D.Polynomial method*), 351
- estimate_parameters() (*hyperspy.api.model.components1D.PowerLaw method*), 352
- estimate_parameters() (*hyperspy.api.model.components1D.SkewNormal method*), 356
- estimate_parameters() (*hyperspy.api.model.components1D.SplitVoigt method*), 358
- estimate_parameters() (*hyperspy.api.model.components1D.Voigt method*), 360
- estimate_peak_width() (*hyperspy.api.signals.Signal1D method*), 467
- estimate_poissonian_noise_variance() (*hyperspy.api.signals.BaseSignal method*), 411
- estimate_shift1D() (*hyperspy.api.signals.Signal1D method*), 467
- estimate_shift2D() (*hyperspy.api.signals.Signal2D method*), 480
- Event (*class in hyperspy.events*), 497
- Events (*class in hyperspy.events*), 500
- EventSuppressor (*class in hyperspy.events*), 499
- Exponential (*class in hyperspy.api.model.components1D*), 341
- export() (*hyperspy.component.Component method*), 533
- export() (*hyperspy.component.Parameter method*), 537
- export() (*hyperspy.misc.utils.DictionaryTreeBrowser method*), 566
- export_bss_results() (*hyperspy.api.signals.BaseSignal method*), 412
- export_cluster_results() (*hyperspy.api.signals.BaseSignal method*), 413
- export_decomposition_results() (*hyperspy.api.signals.BaseSignal method*), 414
- export_results() (*hyperspy.model.BaseModel method*), 516
- export_to_dictionary() (*in module hyperspy.misc.export_dictionary*), 568
- Expression (*class in hyperspy.api.model.components1D*), 342
- ext_bounded (*hyperspy.component.Parameter property*), 538
- ext_force_positive (*hyperspy.component.Parameter property*), 538
- extend() (*hyperspy.model.BaseModel method*), 516

`extend()` (*hyperspy.samfire.Samfire* method), 540

F

`fetch()` (*hyperspy.component.Parameter* method), 538

`fetch_stored_values()` (*hyperspy.model.BaseModel* method), 516

`fetch_values_from_array()` (*hyperspy.component.Component* method), 534

`fetch_values_from_array()` (*hyperspy.model.BaseModel* method), 516

`fft()` (*hyperspy.api.signals.BaseSignal* method), 416

`filter_butterworth()` (*hyperspy.api.signals.Signal1D* method), 468

`find_local_max()` (in module *hyperspy.utils.peakfinders2D*), 569

`find_peaks()` (*hyperspy.api.signals.Signal2D* method), 481

`find_peaks1D_ohaver()` (*hyperspy.api.signals.Signal1D* method), 468

`find_peaks_dog()` (in module *hyperspy.utils.peakfinders2D*), 569

`find_peaks_log()` (in module *hyperspy.utils.peakfinders2D*), 570

`find_peaks_max()` (in module *hyperspy.utils.peakfinders2D*), 570

`find_peaks_minmax()` (in module *hyperspy.utils.peakfinders2D*), 570

`find_peaks_stat()` (in module *hyperspy.utils.peakfinders2D*), 571

`find_peaks_xc()` (in module *hyperspy.utils.peakfinders2D*), 572

`find_peaks_zaefferer()` (in module *hyperspy.utils.peakfinders2D*), 572

`finish()` (*hyperspy.learn.ornmf.ORNMF* method), 503

`finish()` (*hyperspy.learn.rpca.ORPCA* method), 506

`fit()` (*hyperspy.learn.ornmf.ORNMF* method), 503

`fit()` (*hyperspy.learn.rpca.ORPCA* method), 506

`fit()` (*hyperspy.model.BaseModel* method), 517

`fit_component()` (*hyperspy.models.model1d.Model1D* method), 528

`fold()` (*hyperspy.api.signals.BaseSignal* method), 417

`free` (*hyperspy.component.Parameter* attribute), 536

`free_parameters` (*hyperspy.component.Component* property), 534

`from_signal()` (*hyperspy.api.plot.markers.Markers* class method), 368

`function()` (*hyperspy.api.model.components1D.SplitVoigt* method), 358

`function_nd()` (*hyperspy.api.model.components1D.Expression* method), 344

`function_nd()` (*hyperspy.api.model.components1D.Offset* method), 351

`function_nd()` (*hyperspy.api.model.components1D.ScalableFixedPattern* method), 354

`function_nd()` (*hyperspy.api.model.components1D.SplitVoigt* method), 359

`FunctionalDataAxis` (class in *hyperspy.axes*), 493

G

`Gaussian` (class in *hyperspy.api.model.components1D*), 344

`Gaussian2D` (class in *hyperspy.api.model.components2D*), 361

`gaussian_filter()` (*hyperspy.api.signals.Signal1D* method), 469

`GaussianHF` (class in *hyperspy.api.model.components1D*), 345

`generate_values()` (*hyperspy.samfire.Samfire* method), 540

`get_bss_factors()` (*hyperspy.api.signals.BaseSignal* method), 417

`get_bss_loadings()` (*hyperspy.api.signals.BaseSignal* method), 417

`get_bss_model()` (*hyperspy.api.signals.BaseSignal* method), 417

`get_chunk_size()` (*hyperspy._signals.lazy.LazySignal* method), 553

`get_cluster_distances()` (*hyperspy.api.signals.BaseSignal* method), 417

`get_cluster_labels()` (*hyperspy.api.signals.BaseSignal* method), 418

`get_cluster_signals()` (*hyperspy.api.signals.BaseSignal* method), 418

`get_configuration_directory_path()` (in module *hyperspy.api*), 327

`get_current_kwargs()` (*hyperspy.api.plot.markers.HorizontalLines* method), 365

`get_current_kwargs()` (*hyperspy.api.plot.markers.Markers* method), 369

`get_current_kwargs()` (*hyperspy.api.plot.markers.VerticalLines* method), 374

`get_current_signal()` (*hyperspy.api.signals.BaseSignal* method), 418

`get_decomposition_factors()` (*hyperspy.api.signals.BaseSignal* method), 419

`get_decomposition_loadings()` (*hyperspy.api.signals.BaseSignal* method), 419

`get_decomposition_model()` (*hyperspy.api.signals.BaseSignal* method), 419

`get_dimensions_from_data()` (*hyperspy.api.signals.BaseSignal* method), 420

- `get_explained_variance_ratio()` (*hyperspy.api.signals.BaseSignal* method), 420
`get_histogram()` (*hyperspy._signals.LazySignal* method), 553
`get_histogram()` (*hyperspy.api.signals.BaseSignal* method), 420
`get_item()` (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 566
`get_noise_variance()` (*hyperspy.api.signals.BaseSignal* method), 422
`GlobalStrategy` (class in *hyperspy.api.samfire.global_strategies*), 389
`grad_a()` (*hyperspy.api.model.components1D.Bleasdale* method), 339
`grad_b()` (*hyperspy.api.model.components1D.Bleasdale* method), 339
`grad_c()` (*hyperspy.api.model.components1D.Bleasdale* method), 339
`gui()` (*hyperspy.api.model.components1D.ScalableFixedParameter* method), 354
`gui()` (*hyperspy.api.roi.CircleROI* method), 383
`gui()` (*hyperspy.api.roi.Line2DROI* method), 384
`gui()` (*hyperspy.api.roi.Point1DROI* method), 385
`gui()` (*hyperspy.api.roi.Point2DROI* method), 386
`gui()` (*hyperspy.api.roi.RectangularROI* method), 386
`gui()` (*hyperspy.api.roi.SpanROI* method), 387
`gui()` (*hyperspy.axes.AxesManager* method), 488
`gui()` (*hyperspy.axes.BaseDataAxis* method), 490
`gui()` (*hyperspy.component.Component* method), 534
`gui()` (*hyperspy.component.Parameter* method), 538
`gui()` (*hyperspy.model.BaseModel* method), 519
`gui_navigation_sliders()` (*hyperspy.axes.AxesManager* method), 488
- ## H
- `hanning_taper()` (*hyperspy.api.signals.Signal1D* method), 470
`has_item()` (*hyperspy.misc.utils.DictionaryTreeBrowser* method), 567
`has_pool` (*hyperspy.utils.parallel_pool.ParallelPool* property), 542
`HeavisideStep` (class in *hyperspy.api.model.components1D*), 347
`height` (*hyperspy.api.roi.RectangularROI* property), 386
`HistogramStrategy` (class in *hyperspy.api.samfire.global_strategies*), 389
`HorizontalLines` (class in *hyperspy.api.plot.markers*), 364
`hyperspy.api` module, 326
`hyperspy.api.data` module, 334
`hyperspy.api.model` module, 336
`hyperspy.api.model.components1D` module, 337
`hyperspy.api.model.components2D` module, 361
`hyperspy.api.plot` module, 375
`hyperspy.api.plot.markers` module, 362
`hyperspy.api.roi` module, 382
`hyperspy.api.samfire` module, 393
`hyperspy.api.samfire.fit_tests` module, 388
`hyperspy.api.samfire.global_strategies` module, 389
`hyperspy.api.samfire.local_strategies` module, 390
`hyperspy.api.signals` module, 395
`hyperspy.axes` module, 485
`hyperspy.events` module, 497
`hyperspy.learn.mlpca` module, 501
`hyperspy.learn.mva` module, 500
`hyperspy.learn.ornmf` module, 502
`hyperspy.learn.orthomax` module, 505
`hyperspy.learn.rpca` module, 505
`hyperspy.learn.svd_pca` module, 509
`hyperspy.learn.whitening` module, 511
`hyperspy.models.model1d` module, 526
`hyperspy.models.model2d` module, 531
`hyperspy.roi` module, 562
`hyperspy.utils.peakfinders2D` module, 569
- ## I
- `ifft()` (*hyperspy.api.signals.BaseSignal* method), 422
`imag` (*hyperspy.api.signals.ComplexSignal* property), 458
`inav` (*hyperspy.api.signals.BaseSignal* attribute), 396
`indexmax()` (*hyperspy.api.signals.BaseSignal* method), 423

- indexmin()** (*hyperspy.api.signals.BaseSignal method*), 423
indices (*hyperspy.axes.AxesManager property*), 488
init_parameters() (*hyperspy.component.Component method*), 534
insert() (*hyperspy.model.BaseModel method*), 519
integral_as_signal() (*hyperspy.api.model.components1D.GaussianHF method*), 347
integrate1D() (*hyperspy.api.signals.BaseSignal method*), 424
integrate_simpson() (*hyperspy._signals.lazy.LazySignal method*), 555
integrate_simpson() (*hyperspy.api.signals.BaseSignal method*), 425
interactive() (*hyperspy.roi.BaseInteractiveROI method*), 563
interactive() (*in module hyperspy.api*), 327
interpolate_in_between() (*hyperspy.api.signals.Signal1D method*), 470
interpolate_on_axis() (*hyperspy.api.signals.BaseSignal method*), 425
is_ipyparallel (*hyperspy.utils.parallel_pool.ParallelPool property*), 542
is_multiprocessing (*hyperspy.utils.parallel_pool.ParallelPool property*), 542
is_rgb (*hyperspy.api.signals.BaseSignal property*), 426
is_rgba (*hyperspy.api.signals.BaseSignal property*), 426
is_rgbx (*hyperspy.api.signals.BaseSignal property*), 426
is_valid() (*hyperspy.api.roi.CircleROI method*), 383
is_valid() (*hyperspy.api.roi.RectangularROI method*), 386
is_valid() (*hyperspy.api.roi.SpanROI method*), 387
is_valid() (*hyperspy.roi.BaseROI method*), 564
isig (*hyperspy.api.signals.BaseSignal attribute*), 396
iterpath (*hyperspy.axes.AxesManager property*), 488
- K**
- key_navigator()** (*hyperspy.axes.AxesManager method*), 488
keys() (*hyperspy.misc.utils.DictionaryTreeBrowser method*), 568
- L**
- LazyComplexSignal** (*class in hyperspy._lazy_signals*), 546
LazyComplexSignal1D (*class in hyperspy._lazy_signals*), 546
LazyComplexSignal2D (*class in hyperspy._lazy_signals*), 547
LazySignal (*class in hyperspy._signals.lazy*), 548
LazySignal1D (*class in hyperspy._lazy_signals*), 560
LazySignal2D (*class in hyperspy._lazy_signals*), 561
LearningResults (*class in hyperspy.learn.mva*), 500
Line2DROI (*class in hyperspy.api.roi*), 383
Lines (*class in hyperspy.api.plot.markers*), 365
load() (*hyperspy.learn.mva.LearningResults method*), 501
load() (*in module hyperspy.api*), 328
load_parameters_from_file() (*hyperspy.model.BaseModel method*), 519
LocalStrategy (*class in hyperspy.api.samfire.local_strategies*), 390
log() (*hyperspy.samfire.Samfire method*), 540
Logistic (*class in hyperspy.api.model.components1D*), 348
Lorentzian (*class in hyperspy.api.model.components1D*), 349
luminescence_signal() (*in module hyperspy.api.data*), 335
- M**
- map** (*hyperspy.component.Parameter attribute*), 536
map() (*hyperspy.api.samfire.fit_tests.AIC_test method*), 388
map() (*hyperspy.api.samfire.fit_tests.AICc_test method*), 388
map() (*hyperspy.api.samfire.fit_tests.BIC_test method*), 388
map() (*hyperspy.api.samfire.fit_tests.red_chisq_test method*), 388
map() (*hyperspy.api.signals.BaseSignal method*), 426
Markers (*class in hyperspy.api.plot.markers*), 366
max() (*hyperspy.api.signals.BaseSignal method*), 428
mean (*hyperspy.api.model.components1D.SkewNormal property*), 357
mean() (*hyperspy.api.signals.BaseSignal method*), 429
metadata (*hyperspy.api.signals.BaseSignal property*), 430
min() (*hyperspy.api.signals.BaseSignal method*), 430
mlpca() (*in module hyperspy.learn.mlpca*), 501
mode (*hyperspy.api.model.components1D.SkewNormal property*), 357
Model1D (*class in hyperspy.models.model1d*), 526
Model2D (*class in hyperspy.models.model2d*), 531
ModelComponents (*class in hyperspy.model*), 525
ModelManager (*class in hyperspy.signal*), 544
module
 hyperspy.api, 326
 hyperspy.api.data, 334
 hyperspy.api.model, 336
 hyperspy.api.model.components1D, 337
 hyperspy.api.model.components2D, 361
 hyperspy.api.plot, 375
 hyperspy.api.plot.markers, 362
 hyperspy.api.roi, 382

hyperspy.api.samfire, 393
 hyperspy.api.samfire.fit_tests, 388
 hyperspy.api.samfire.global_strategies, 389
 hyperspy.api.samfire.local_strategies, 390
 hyperspy.api.signals, 395
 hyperspy.axes, 485
 hyperspy.events, 497
 hyperspy.learn.mlpca, 501
 hyperspy.learn.mva, 500
 hyperspy.learn.ornmf, 502
 hyperspy.learn.orthomax, 505
 hyperspy.learn.rpca, 505
 hyperspy.learn.svd_pca, 509
 hyperspy.learn.whitening, 511
 hyperspy.models.model1d, 526
 hyperspy.models.model2d, 531
 hyperspy.roi, 562
 hyperspy.utils.peakfinders2D, 569
 multifit() (*hyperspy.model.BaseModel* method), 519

N

nanmax() (*hyperspy.api.signals.BaseSignal* method), 430
 nanmean() (*hyperspy.api.signals.BaseSignal* method), 431
 nanmin() (*hyperspy.api.signals.BaseSignal* method), 431
 nanstd() (*hyperspy.api.signals.BaseSignal* method), 431
 nansum() (*hyperspy.api.signals.BaseSignal* method), 431
 nanvar() (*hyperspy.api.signals.BaseSignal* method), 431
 navigation_axes (*hyperspy.axes.AxesManager* property), 488
 navigation_dimension (*hyperspy.axes.AxesManager* property), 488
 navigation_extent (*hyperspy.axes.AxesManager* property), 488
 navigation_shape (*hyperspy.axes.AxesManager* property), 489
 navigation_size (*hyperspy.axes.AxesManager* property), 489
 need_pixels (*hyperspy.api.samfire.SamfirePool* property), 394
 normalize_bss_components() (*hyperspy.api.signals.BaseSignal* method), 431
 normalize_decomposition_components() (*hyperspy.api.signals.BaseSignal* method), 431
 normalize_poissonian_noise() (*hyperspy.api.signals.BaseSignal* method), 431

O

Offset (*class in hyperspy.api.model.components1D*), 350
 offset_transform (*hyperspy.api.plot.markers.Markers* property), 369
 original_metadata (*hyperspy.api.signals.BaseSignal* property), 432
 ORNMF (*class in hyperspy.learn.ornmf*), 502
 ornmf() (*in module hyperspy.learn.ornmf*), 504
 ORPCA (*class in hyperspy.learn.rpca*), 505
 orpca() (*in module hyperspy.learn.rpca*), 507
 orthomax() (*in module hyperspy.learn.orthomax*), 505

P

ParallelPool (*class in hyperspy.utils.parallel_pool*), 541
 Parameter (*class in hyperspy.component*), 536
 parameters (*hyperspy.component.Component* property), 534
 parse() (*hyperspy.api.samfire.SamfirePool* method), 394
 phase (*hyperspy.api.signals.ComplexSignal* property), 458
 ping_workers() (*hyperspy.api.samfire.SamfirePool* method), 394
 pixels_done (*hyperspy.samfire.Samfire* property), 541
 pixels_left (*hyperspy.samfire.Samfire* property), 541
 plot() (*hyperspy._signals.lazy.LazySignal* method), 556
 plot() (*hyperspy.api.plot.markers.Markers* method), 369
 plot() (*hyperspy.api.samfire.global_strategies.GlobalStrategy* method), 389
 plot() (*hyperspy.api.samfire.global_strategies.HistogramStrategy* method), 389
 plot() (*hyperspy.api.samfire.local_strategies.LocalStrategy* method), 390
 plot() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy* method), 391
 plot() (*hyperspy.api.signals.BaseSignal* method), 432
 plot() (*hyperspy.api.signals.ComplexSignal* method), 458
 plot() (*hyperspy.api.signals.ComplexSignal2D* method), 461
 plot() (*hyperspy.api.signals.Signal1D* method), 470
 plot() (*hyperspy.api.signals.Signal2D* method), 483
 plot() (*hyperspy.component.Component* method), 534
 plot() (*hyperspy.component.Parameter* method), 538
 plot() (*hyperspy.models.model1d.Model1D* method), 529
 plot() (*hyperspy.samfire.Samfire* method), 541
 plot_bss_factors() (*hyperspy.api.signals.BaseSignal* method), 433
 plot_bss_loadings() (*hyperspy.api.signals.BaseSignal* method), 434

- plot_bss_results() (*hyperspy.api.signals.BaseSignal method*), 434
 plot_cluster_distances() (*hyperspy.api.signals.BaseSignal method*), 435
 plot_cluster_labels() (*hyperspy.api.signals.BaseSignal method*), 436
 plot_cluster_metric() (*hyperspy.api.signals.BaseSignal method*), 437
 plot_cluster_results() (*hyperspy.api.signals.BaseSignal method*), 437
 plot_cluster_signals() (*hyperspy.api.signals.BaseSignal method*), 437
 plot_colorbar() (*hyperspy.api.plot.markers.Markers method*), 369
 plot_cumulative_explained_variance_ratio() (*hyperspy.api.signals.BaseSignal method*), 438
 plot_decomposition_factors() (*hyperspy.api.signals.BaseSignal method*), 438
 plot_decomposition_loadings() (*hyperspy.api.signals.BaseSignal method*), 439
 plot_decomposition_results() (*hyperspy.api.signals.BaseSignal method*), 439
 plot_explained_variance_ratio() (*hyperspy.api.signals.BaseSignal method*), 440
 plot_histograms() (*in module hyperspy.api.plot*), 375
 plot_images() (*in module hyperspy.api.plot*), 376
 plot_results() (*hyperspy.model.BaseModel method*), 520
 plot_roi_map() (*in module hyperspy.api.plot*), 379
 plot_signals() (*in module hyperspy.api.plot*), 380
 plot_spectra() (*in module hyperspy.api.plot*), 381
 Point1DROI (*class in hyperspy.api.roi*), 385
 Point2DROI (*class in hyperspy.api.roi*), 385
 Points (*class in hyperspy.api.plot.markers*), 371
 Polygons (*class in hyperspy.api.plot.markers*), 371
 Polynomial (*class in hyperspy.api.model.components1D*), 351
 pop() (*hyperspy.signal.ModelManager method*), 544
 PowerLaw (*class in hyperspy.api.model.components1D*), 351
 prepare_interpolator() (*hyperspy.api.model.components1D.ScalableFixedPattern method*), 355
 prepare_workers() (*hyperspy.api.samfire.SamfirePool method*), 394
 print_current_values() (*hyperspy.component.Component method*), 534
 print_current_values() (*hyperspy.model.BaseModel method*), 521
 print_known_signal_types() (*in module hyperspy.api*), 332
 print_summary_statistics() (*hyperspy.api.signals.BaseSignal method*), 442
 process_lazy_attributes() (*hyperspy.misc.utils.DictionaryTreeBrowser method*), 568
 profile_line() (*hyperspy.api.roi.Line2DROI static method*), 384
 project() (*hyperspy.learn.ornmf.ORNMF method*), 503
 project() (*hyperspy.learn.rpca.ORPCA method*), 507
- ## R
- radii (*hyperspy.api.samfire.local_strategies.LocalStrategy property*), 390
 radii (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy property*), 392
 ragged (*hyperspy.api.signals.BaseSignal property*), 442
 RC (*class in hyperspy.api.model.components1D*), 353
 real (*hyperspy.api.signals.ComplexSignal property*), 459
 rebin() (*hyperspy._signals.lazy.LazySignal method*), 557
 rebin() (*hyperspy.api.signals.BaseSignal method*), 442
 rechunk() (*hyperspy._signals.lazy.LazySignal method*), 559
 Rectangles (*class in hyperspy.api.plot.markers*), 372
 RectangularROI (*class in hyperspy.api.roi*), 386
 red_chisq (*hyperspy.model.BaseModel property*), 521
 red_chisq_test (*class in hyperspy.api.samfire.fit_tests*), 388
 ReducedChiSquaredStrategy (*class in hyperspy.api.samfire.local_strategies*), 391
 refresh() (*hyperspy.api.samfire.global_strategies.GlobalStrategy method*), 389
 refresh() (*hyperspy.api.samfire.global_strategies.HistogramStrategy method*), 390
 refresh() (*hyperspy.api.samfire.local_strategies.LocalStrategy method*), 390
 refresh() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy method*), 392
 refresh_database() (*hyperspy.samfire.Samfire method*), 541
 remove() (*hyperspy.api.samfire.global_strategies.GlobalStrategy method*), 389
 remove() (*hyperspy.api.samfire.global_strategies.HistogramStrategy method*), 390
 remove() (*hyperspy.api.samfire.local_strategies.LocalStrategy method*), 391
 remove() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy method*), 392
 remove() (*hyperspy.axes.AxesManager method*), 489
 remove() (*hyperspy.model.BaseModel method*), 521
 remove() (*hyperspy.models.model1d.Model1D method*), 530
 remove() (*hyperspy.samfire.Samfire method*), 541
 remove() (*hyperspy.signal.ModelManager method*), 544
 remove_background() (*hyperspy.api.signals.Signal1D method*), 472

- remove_items() (*hyperspy.api.plot.markers.Markers method*), 370
- remove_signal_range() (*hyperspy.models.model1d.Model1D method*), 530
- remove_signal_range() (*hyperspy.models.model2d.Model2D method*), 532
- remove_widget() (*hyperspy.roi.BaseInteractiveROI method*), 564
- reset_signal_range() (*hyperspy.models.model1d.Model1D method*), 530
- reset_signal_range() (*hyperspy.models.model2d.Model2D method*), 532
- restore() (*hyperspy.signal.ModelManager method*), 544
- reverse_bss_component() (*hyperspy.api.signals.BaseSignal method*), 443
- reverse_decomposition_component() (*hyperspy.api.signals.BaseSignal method*), 444
- rollaxis() (*hyperspy.api.signals.BaseSignal method*), 444
- rpca_godec() (*in module hyperspy.learn.rpca*), 508
- run() (*hyperspy.api.samfire.SamfirePool method*), 394
- ## S
- samf (*hyperspy.api.samfire.local_strategies.LocalStrategy property*), 391
- samf (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy property*), 392
- Samfire (*class in hyperspy.samfire*), 539
- SamfirePool (*class in hyperspy.api.samfire*), 393
- save() (*hyperspy.api.signals.BaseSignal method*), 445
- save() (*hyperspy.learn.mva.LearningResults method*), 501
- save() (*hyperspy.model.BaseModel method*), 521
- save_parameters2file() (*hyperspy.model.BaseModel method*), 521
- ScalableFixedPattern (*class in hyperspy.api.model.components1D*), 353
- set_axis() (*hyperspy.axes.AxesManager method*), 489
- set_component_active_value() (*hyperspy.model.BaseModel method*), 522
- set_item() (*hyperspy.misc.utils.DictionaryTreeBrowser method*), 568
- set_log_level() (*in module hyperspy.api*), 332
- set_noise_variance() (*hyperspy.api.signals.BaseSignal method*), 446
- set_parameters_free() (*hyperspy.component.Component method*), 535
- set_parameters_free() (*hyperspy.model.BaseModel method*), 522
- set_parameters_not_free() (*hyperspy.component.Component method*), 535
- set_parameters_not_free() (*hyperspy.model.BaseModel method*), 523
- set_parameters_value() (*hyperspy.model.BaseModel method*), 524
- set_ScalarMappable_array() (*hyperspy.api.plot.markers.Markers method*), 370
- set_signal_origin() (*hyperspy.api.signals.BaseSignal method*), 447
- set_signal_range() (*hyperspy.models.model1d.Model1D method*), 530
- set_signal_range() (*hyperspy.models.model2d.Model2D method*), 532
- set_signal_range_from_mask() (*hyperspy.model.BaseModel method*), 524
- set_signal_type() (*hyperspy.api.signals.BaseSignal method*), 447
- setup() (*hyperspy.utils.parallel_pool.ParallelPool method*), 542
- shift1D() (*hyperspy.api.signals.Signal1D method*), 473
- signal (*hyperspy.model.BaseModel property*), 525
- Signal1D (*class in hyperspy.api.signals*), 464
- Signal2D (*class in hyperspy.api.signals*), 477
- signal_axes (*hyperspy.axes.AxesManager property*), 489
- signal_dimension (*hyperspy.axes.AxesManager property*), 489
- signal_extent (*hyperspy.axes.AxesManager property*), 489
- signal_shape (*hyperspy.axes.AxesManager property*), 489
- signal_size (*hyperspy.axes.AxesManager property*), 489
- skewness (*hyperspy.api.model.components1D.SkewNormal property*), 357
- SkewNormal (*class in hyperspy.api.model.components1D*), 355
- sleep() (*hyperspy.utils.parallel_pool.ParallelPool method*), 542
- smooth_lowess() (*hyperspy.api.signals.Signal1D method*), 474
- smooth_savitzky_golay() (*hyperspy.api.signals.Signal1D method*), 475
- smooth_tv() (*hyperspy.api.signals.Signal1D method*), 475
- SpanROI (*class in hyperspy.api.roi*), 387
- spikes_diagnosis() (*hyperspy.api.signals.Signal1D method*), 476
- spikes_removal_tool() (*hyperspy.api.signals.Signal1D method*), 476
- split() (*hyperspy.api.signals.BaseSignal method*), 448

- SplitVoigt (class in *hyperspy.api.model.components1D*), 357
- Squares (class in *hyperspy.api.plot.markers*), 373
- squeeze() (*hyperspy.api.signals.BaseSignal* method), 449
- stack() (in module *hyperspy.api*), 333
- start() (*hyperspy.samfire.Samfire* method), 541
- std() (*hyperspy.api.signals.BaseSignal* method), 449
- stop() (*hyperspy.api.samfire.SamfirePool* method), 394
- stop() (*hyperspy.samfire.Samfire* method), 541
- store() (*hyperspy.model.BaseModel* method), 525
- store() (*hyperspy.signal.ModelManager* method), 544
- store_current_value_in_array() (*hyperspy.component.Parameter* method), 538
- store_current_values() (*hyperspy.model.BaseModel* method), 525
- sum() (*hyperspy.api.signals.BaseSignal* method), 450
- summary() (*hyperspy.learn.mva.LearningResults* method), 501
- suppress() (*hyperspy.events.Event* method), 498
- suppress() (*hyperspy.events.Events* method), 500
- suppress() (*hyperspy.events.EventSuppressor* method), 500
- suppress_callback() (*hyperspy.events.Event* method), 498
- suspend_update() (*hyperspy.model.BaseModel* method), 525
- svd_flip_signs() (in module *hyperspy.learn.svd_pca*), 509
- svd_pca() (in module *hyperspy.learn.svd_pca*), 509
- svd_solve() (in module *hyperspy.learn.svd_pca*), 510
- swap_axes() (*hyperspy.api.signals.BaseSignal* method), 451
- switch_iterpath() (*hyperspy.axes.AxesManager* method), 489
- ## T
- T (*hyperspy.api.signals.BaseSignal* property), 397
- test() (*hyperspy.api.samfire.fit_tests.AIC_test* method), 388
- test() (*hyperspy.api.samfire.fit_tests.AICc_test* method), 388
- test() (*hyperspy.api.samfire.fit_tests.BIC_test* method), 388
- test() (*hyperspy.api.samfire.fit_tests.red_chisq_test* method), 388
- Texts (class in *hyperspy.api.plot.markers*), 373
- to_device() (*hyperspy.api.signals.BaseSignal* method), 451
- to_host() (*hyperspy.api.signals.BaseSignal* method), 452
- to_signal1D() (*hyperspy._signals.common_signal1d.CommonSignal1D* method), 545
- to_signal2D() (*hyperspy._signals.common_signal1d.CommonSignal1D* method), 545
- tolerance (*hyperspy.api.samfire.fit_tests.red_chisq_test* property), 388
- transform (*hyperspy.api.plot.markers.Markers* property), 371
- transpose() (*hyperspy.api.signals.BaseSignal* method), 452
- transpose() (in module *hyperspy.api*), 334
- trigger() (*hyperspy.events.Event* method), 499
- twin (*hyperspy.component.Parameter* property), 539
- twin_function_expr (*hyperspy.component.Parameter* property), 539
- twin_inverse_function_expr (*hyperspy.component.Parameter* property), 539
- two_gaussians() (in module *hyperspy.api.data*), 336
- ## U
- undo_treatments() (*hyperspy.api.signals.BaseSignal* method), 453
- unfold() (*hyperspy.api.signals.BaseSignal* method), 453
- unfold_navigation_space() (*hyperspy.api.signals.BaseSignal* method), 453
- unfold_signal_space() (*hyperspy.api.signals.BaseSignal* method), 453
- unfolded() (*hyperspy.api.signals.BaseSignal* method), 454
- UniformDataAxis (class in *hyperspy.axes*), 494
- UnitConversion (class in *hyperspy.axes*), 496
- unwrapped_phase() (*hyperspy.api.signals.ComplexSignal* method), 459
- update() (*hyperspy.api.plot.markers.Arrows* method), 363
- update() (*hyperspy.api.plot.markers.Markers* method), 371
- update() (*hyperspy.api.samfire.global_strategies.GlobalStrategy* method), 389
- update() (*hyperspy.api.samfire.global_strategies.HistogramStrategy* method), 390
- update() (*hyperspy.api.samfire.local_strategies.LocalStrategy* method), 391
- update() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy* method), 392
- update() (*hyperspy.roi.BaseInteractiveROI* method), 564
- update() (*hyperspy.roi.BaseROI* method), 564
- update() (*hyperspy.samfire.Samfire* method), 541
- update_axes_attributes_from() (*hyperspy.axes.AxesManager* method), 490
- update_axis() (*hyperspy.axes.DataAxis* method), 492

update_from() (*hyperspy.axes.BaseDataAxis* method), 491
 update_from() (*hyperspy.axes.DataAxis* method), 492
 update_from() (*hyperspy.axes.FunctionalDataAxis* method), 494
 update_from() (*hyperspy.axes.UniformDataAxis* method), 495
 update_parameters() (*hyperspy.api.samfire.SamfirePool* method), 394
 update_plot() (*hyperspy.api.signals.BaseSignal* method), 454
 update_plot() (*hyperspy.model.BaseModel* method), 525

V

value (*hyperspy.component.Parameter* attribute), 536
 value2index() (*hyperspy.axes.BaseDataAxis* method), 491
 value2index() (*hyperspy.axes.UniformDataAxis* method), 495
 value_range_to_indices() (*hyperspy.axes.BaseDataAxis* method), 491
 valuemax() (*hyperspy._signals.lazy.LazySignal* method), 559
 valuemax() (*hyperspy.api.signals.BaseSignal* method), 454
 valuemin() (*hyperspy._signals.lazy.LazySignal* method), 560
 valuemin() (*hyperspy.api.signals.BaseSignal* method), 455
 values() (*hyperspy.api.samfire.global_strategies.GlobalStrategy* method), 389
 values() (*hyperspy.api.samfire.global_strategies.HistogramStrategy* method), 390
 values() (*hyperspy.api.samfire.local_strategies.LocalStrategy* method), 391
 values() (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy* method), 392
 var() (*hyperspy.api.signals.BaseSignal* method), 455
 variance (*hyperspy.api.model.components1D.SkewNormal* property), 357
 VerticalLines (class in *hyperspy.api.plot.markers*), 374
 Voigt (class in *hyperspy.api.model.components1D*), 359

W

wave_image() (in module *hyperspy.api.data*), 336
 weight (*hyperspy.api.samfire.local_strategies.LocalStrategy* property), 391
 weight (*hyperspy.api.samfire.local_strategies.ReducedChiSquaredStrategy* property), 392
 whiten_data() (in module *hyperspy.learn.whitening*), 511
 width (*hyperspy.api.roi.RectangularROI* property), 387